Oracle® Application Development Framework Tutorial (10.1.3.1.0)

September, 2006

Oracle Application Development Framework: Tutorial (10.1.3.1.0)

Primary Authors: Jeff Gallus, Gary Williams, Kate Heap

Contributors: Lynn Munsinger, Duncan Mills, Frank Nimphius

Preface

This preface outlines the contents and audience for the Oracle Application Development Framework: Tutorial (10.1.3.1.0).

The preface contains the following sections:

- Intended Audience
- Structure
- Related Documents

Intended Audience

This tutorial is for J2EE developers who use Oracle Application Development Framework (ADF) to build Web applications.

Structure

The tutorial consists of the following chapters:

Chapter 1: "Getting Started"

This chapter describes the Service Request scenario and installation of the schema.

Chapter 2: "Developing the Data Model"

This chapter describes how to build the data model for your application by using Oracle TopLink and Oracle ADF.

Chapter 3: "Defining the Page Flow and Navigation"

This chapter describes how to create the skeleton pages in your JSF application and define the navigation between them.

Chapter 4: "Developing Application Standards"

This chapter discusses the role of standards in application development and demonstrates how to implement them in the SRDemo application.

Chapter 5: "Developing a Simple Display Page"

This chapter describes how to create a simple display page at the center of the SRDemo application that enables users to view information about service requests.

Chapter 6: "Implementing Login Security"

This chapter describes how to build security for the SRDemo application.

Chapter 7: "Developing a Search Page"

This chapter describes how to build a search page. The page contains two sections, one used to specify the query criteria and the other to display the results.

Chapter 8: "Developing a Master-Detail Page"

In this chapter, you develop a master-detail page that shows a service request and its Service Request History rows.

Chapter 9: "Implementing Transactional Capabilities"

This chapter describes how to build the pages to create a service request. The service request process involves three main pages: one to specify the product and problem, one to confirm the values, and one to commit and display the service request ID. You also create a fourth page, which displays some frequently asked questions about solving some typical product problems.

Chapter 10: "Developing an Edit Page"

This chapter describes how to create a page that enables managers and technicians to edit service requests.

Chapter 11: "Deploying the Application to Oracle Application Server 10*g*"

In this chapter, you use JDeveloper to create a deployable package that contains your application and required deployment descriptors. You then deploy the package.

Related Documents

For more information about building applications with Oracle ADF, see the following publications:

- *Oracle Application Development Framework Developer's Guide 10g Release 3 (10.1.3)*
- *Oracle Application Development Framework Developer's Guide for Forms/4GL Developers 10g Release 3 (10.1.3)*

1

Getting Started

This tutorial describes how to build an end-to-end J2EE Web application by using Oracle JDeveloper, Oracle ADF, and Oracle TopLink. The application uses various J2EE technologies, including Enterprise JavaBeans (EJB) and JavaServer Faces (JSF). In the tutorial, you learn how to use JSF for the application's user interface and for control of application navigation.

This chapter contains the following sections:

- Tutorial Scenario: Overview
- Using This Tutorial
- Starting JDeveloper
- Creating a JDeveloper Database Connection
- Defining an Application and its Projects in JDeveloper
- Summary

Tutorial Scenario: Overview

ServiceCompany is a company that provides service support for large household appliances (dishwashers, washing machines, microwaves, and so on). It handles support for a wide variety of appliances and attempts to solve most customer issues by responding to service requests over the Web.

ServiceCompany has found that over time, customers can resolve most issues after they have the correct information. This approach has been shown to save time and money for both the company and its customers. A service request can be created at the request of a customer, technician, or manager.

Service requests opened by employees can represent any type of internal information associated with a product (examples include product recalls, specific problems with products, and so on).

The Business Problem

ServiceCompany has seen service requests increase but resolutions to them have decreased over the past two quarters. As a consequence, the company has decided to implement a more customer-friendly system and a more efficient and speedy request resolution service.

ServiceCompany wants to be able to provide the same service and response to all customers, regardless of the channel of service request placement and service request type.

Business Goals

ServiceCompany has the following goals:

- Record and track product-related service requests
- Resolve service requests smoothly, efficiently, and quickly
- Manage the assignment and record the progress of all service requests
- Completely automate the service-request process
- Enable managers to assign service requests to qualified technicians
- Enable customers and technicians to log service requests
- Track the technicians' areas of product expertise

Business Solution

ServiceCompany has decided to implement a new, fully automated system that is built using Oracle Application Development Framework (ADF). This enables highly productive development of a standards-based J2EE application structure. The application server will be Oracle Application Server 10*g* XE edition.

The major components of the new application are:

- A customer interface to enable any user (customer, technician, or manager) to add, update, and check the status of service requests

- User interfaces with which the company can create, update, and manage service requests. This includes assigning requests to the appropriate technician and gathering cumulative history information.
- Various reporting tools to ensure timely resolution of service requests
- A user interface with which technicians can update their areas of product expertise

The following process represents the planned flow of a customer-generated service request:

1. A customer issues a request via a Web interface.
2. A manager assigns the request to a technician.
3. The technician reviews the request and then either supplies a solution or asks the customer for more information.
4. The customer checks the request and either closes the request or provides further information.
5. Managers can review an existing request for a technician and (if necessary) reassign it to another technician.
6. Technicians identify products in their area of expertise. Managers can then use this information in assigning service requests.

The technologies to be employed in building the application are as follows:

- The technology employed will be Oracle ADF.
- The data will be stored in Oracle Database 10*g*.
- The data model and business logic will be implemented by using Oracle TopLink and Enterprise JavaBeans.
- Databinding (mapping between client components and the business logic) will be provided by Oracle ADF.
- The Web client layer will be built using JSF pages and ADF Faces components.
- Authorization will be based on J2EE container security.
- Deployment of the application will be to Oracle Application Server 10*g* XE edition.

Design Patterns and Architectural Frameworks

A good practice when developing applications is to employ design patterns. Design patterns are a convenient way of reusing object-oriented concepts between applications and between developers. The idea behind design patterns is simple: document and catalog common behavior patterns between objects. Developers can then make use of these patterns rather than re-create them.

In addition to design patterns, developers often use architectural frameworks to build applications that perform in a standard way. One of the frequently used architectural patterns is the Model-View-Controller (MVC) pattern.

In the MVC architecture, the user input, the business logic, and the visual feedback to the

user are explicitly separated and handled by three types of objects. Each of these objects is specialized for a particular role in the application:

- The **view** manages the presentation of the application output to the user.
- The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- The **model** manages the data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

The formal separation of these three components is a key characteristic of a good design.

Another popular design pattern for J2EE applications is the Session Facade pattern. The Session Facade pattern hides complex interactions between the application components from the client's view. It encapsulates the business logic that participates in the application workflow and therefore simplifies the interactions between application components. The session bean (representing the Session Facade) manages the relationships between business objects. The session bean also manages the life cycle of these participants by creating, locating (looking up), modifying, and deleting them as required by the application.

In this tutorial, you use the Session Facade design pattern in the application’s model, and you use the MVC architecture through the use of Oracle ADF and JavaServer Faces.

Using This Tutorial

The tutorial is divided into separate chapters, with each chapter building on the previous one. You must complete each chapter in the order presented in the tutorial.

This section describes all of the prerequisite steps that you need to complete before starting to build the application itself.

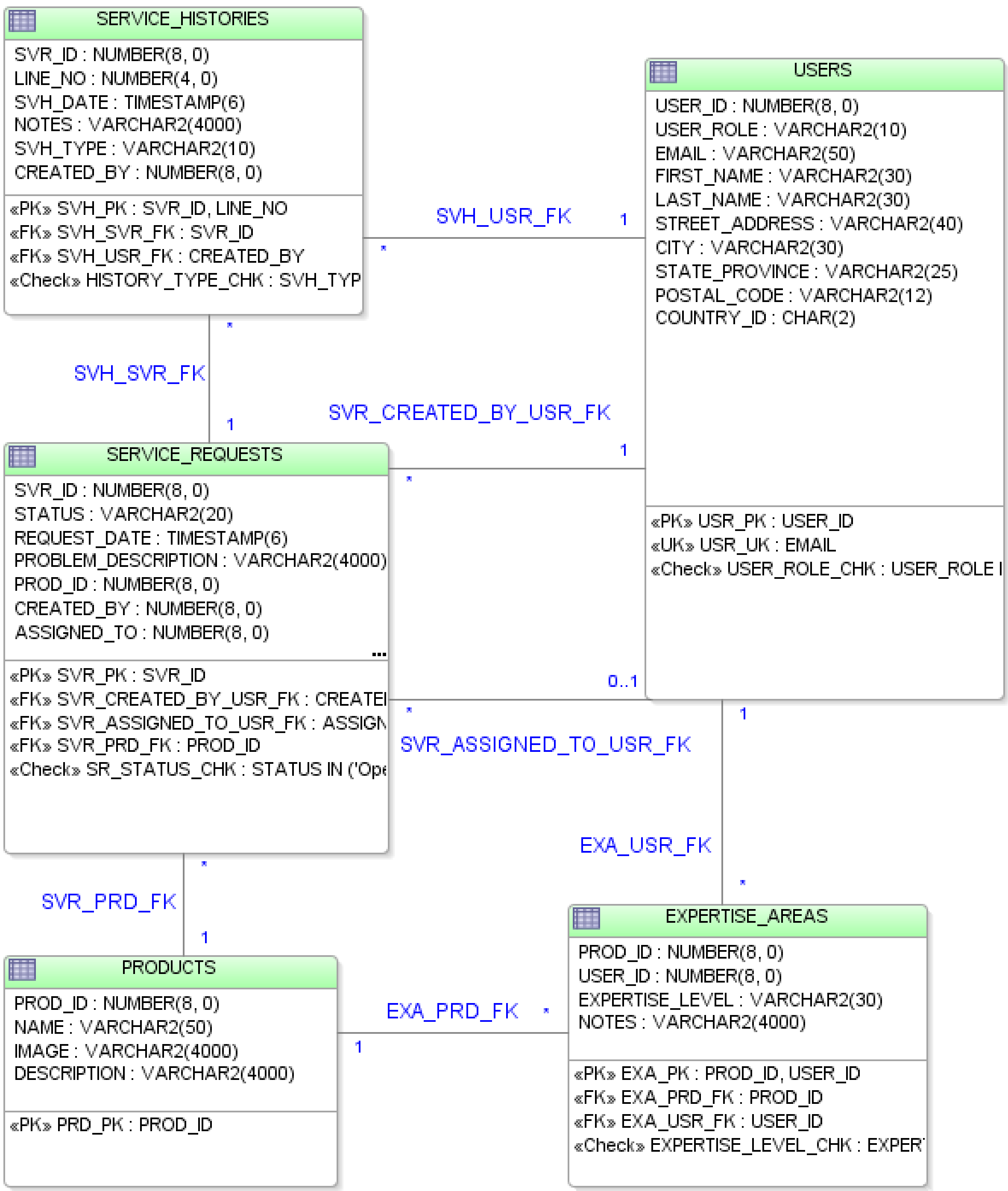
Setting Up Your Environment

You need to prepare your working environment to support the tutorial application. To do this, you perform the following key tasks:

- Prepare to install the schema
- Create the SRDEMO schema owner and install the Service Request schema
- Start Oracle JDeveloper (10.1.3.1.0)
- Create a JDeveloper database connection
- Define an application and projects for the tutorial

Preparing to Install the Schema

The schema consists of five tables and three database sequences. The tables are diagrammed as follows:



The five tables represent creating and assigning a service request to a qualified technician.

Tables

USERS: This table stores all the users who interact with the system, including customers, technicians, and managers. The e-mail address, first and last name, street address, city, state, postal code, and country of each user are stored. An ID uniquely identifies a user.

SERVICE_REQUESTS: This table represents both internal and external requests for activity on a specific product. In all cases, the requests are for a solution to a problem with a product. When a service request is created, the date of the request, the name of the individual who opened it, and the related product are all recorded. A short description of the problem is also stored. After the request is assigned to a technician, the name of the technician and date of assignment are also recorded. An artificial ID uniquely identifies all service requests.

SERVICE_HISTORIES: For each service request, there may be many events recorded. The date the request was created, the name of the individual who created it, and specific notes about the event are all recorded. Any internal communications related to a service request are also tracked. The service request and its sequence number uniquely identify each service history.

PRODUCTS: This table stores all of the products handled by the company. For each product, the name and description are recorded. If an image of the product is available, that too is stored. An artificial ID uniquely identifies all products.

EXPERTISE_AREAS: To better assign technicians to requests, the specific areas of expertise of each technician are defined.

Sequences

USERS_SEQ: Populates the ID for new users

PRODUCTS_SEQ: Populates the ID for each product

SERVICE_REQUESTS_SEQ: Populates the ID for each new service request

1. Obtain the **ADF_tutorial_setup.zip** file at the following location: http://download.oracle.com/otndocs/products/jdev/10131/ADF_tutorial_setup.zip
2. Unzip the file to a temporary directory (e.g., `C:\temp\ADFTutorialSetup\`) to expose the files that are used to create the three images for the Web pages. For the remainder of the tutorial, this directory is referred to as <tutorial_install>.

Installing the Service Request Schema

The `SRDEMO` user owns the data displayed in the application. Access to an Oracle `SYSTEM` user or equivalent is required to create the user account and to assign the appropriate privileges. The `createSchema.sql` file contains all the commands necessary to create the database user. The `createSchemaObjects.sql` file connects as the `SRDEMO` user and creates all the tables, constraints, and database sequences for the tutorial. Finally, the `populateSchemasTables.sql` file inserts example data into the tables for use during the tutorial.

> **Caution:** For security reasons, it is not advisable to install the ADF tutorial schema into a production database. You may need the assistance of your DBA to access an account with the privilege to create a user.

1. Invoke SQL*Plus and log on as SYSTEM or as another DBA-level user. You may need to ask your DBA to give you an account or to run the scripts for you.
2. In the SQL*Plus window, start the `build.sql` script from the directory where you unzipped it. For example:

   ```
   SQLPLUS>Start <tutorial_install>\scripts\build.sql
   ```

   After control is returned to the `build.sql` script, the list of the created objects is displayed along with any potential invalid objects. Running these scripts should take less than 30 seconds. You may rerun the `build.sql` script to drop and re-create the `SRDEMO` owner and objects.

Starting Oracle JDeveloper

Follow these instructions to prepare JDeveloper Studio.

> **Note:** If you have not already installed JDeveloper 10*g* Release 3, then do so before proceeding with the next tutorial steps.

1. In Windows Explorer, navigate to the directory where JDeveloper is installed. In the root directory, double-click the **JDeveloper.exe** icon to invoke JDeveloper. If this is the first time JDeveloper has been run, a "Do you wish to migrate" window appears.
2. Click **No** to continue. You will build the entire application from scratch.
3. On startup, a "Tip of the Day" window appears. These tips are things you can do to make development more productive. Click **Close** when you've finished looking at the tips.

Creating a Database Connection to Access the `SRDEMO` Schema

Follow these instructions to create a new database connection to the Service Request schema using the `SRDEMO` user.

Note: In the tutorial, the database connection is named **`SRDemo`**. The name of the connection does not affect the ability to complete the tutorial. However, we strongly recommend using the naming conventions described in all the steps. In doing so, it is easier to follow the instructions.

---

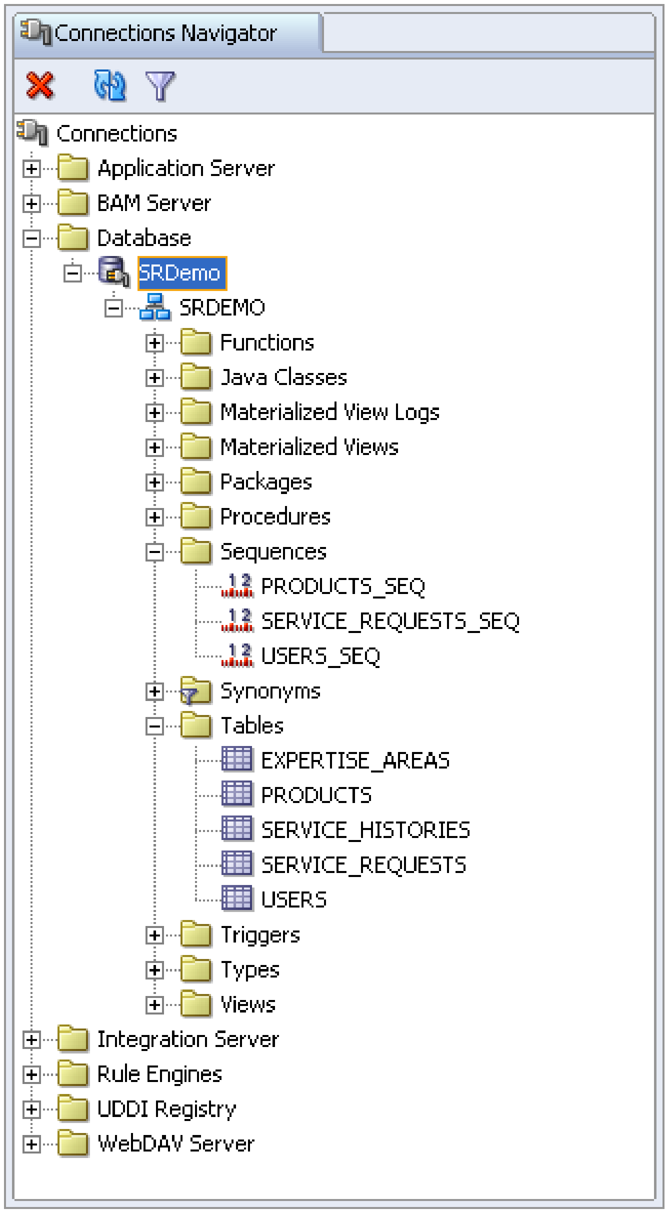
1. In JDeveloper, choose **View > Connections Navigator**.
2. Right-click the **Database** node and choose **New Database Connection**.
3. Click **Next** on the Welcome page.
4. In the Connection Name field, type the connection name **`SRDemo`**. Then click **Next**.
5. On the Authentication page, enter the values as shown in the following table. Then click **Next**.

Field	Value
Username	`SRDEMO`
Password	`Oracle`
Deploy Password	Select the check box

6. On the Connection page, enter the following values. Then click **Next**.

Field	Value
Host Name	`localhost` This is the default host name if the database is on the same machine as JDeveloper. If the database is on anther machine, type the name (or IP address) of the computer where the database is located.
JDBC Port	`1521` This is the default value for the port used to access the database. If you do not know this value, check with your database administrator.
SID	`ORCL` This is the default value for the SID that is used to connect to the database. If you do not know this value, check with your database administrator.

7. Click **Test Connection**. If the database is available and the connection details are correct, then continue. If not, click the **Back** button and check the values.
8. Click **Finish**. The connection now appears below the Database Connection node in the Connections Navigator.
9. You can now examine the schema from JDeveloper. In the Connections Navigator, expand **Database > SRDemo**. Browse the database elements for the schema and confirm that they match the schema definition above.



Defining an Application and Its Projects in JDeveloper

In JDeveloper, you work within projects that are contained in applications.

An application is the highest level in the control structure, serving as a collector of all the subparts of the application. When you open JDeveloper, the applications, which were opened when you last closed JDeveloper, are opened by default.

A JDeveloper project is an organization structure used to logically group related files. In a J2EE application, a project typically represents a part of the application architecture, such as the data model or a part of the client.

You can add multiple projects to your application to easily organize, access, modify, and reuse your source code.

Before you create any application components, you must first create the application and its projects. Use the following procedure to create the new SRDemo application and its projects.

> **Note:** Do not include special characters in the project name (such as periods) or in any activity or element names. If you include special characters, errors appear when you attempt to compile your project.

1. To create an application, click in the **Applications Navigator**, right-click the **Applications** node, and select **New Application** from the shortcut menu.
2. In the Create Application window, enter the following values:

Field	Value
Application Name	`SRDemo`
Directory name	<jdev_install>`\jdev\mywork\SRDemo` Keep the default value. If you used the default directory structure, then your path should match this value. The directory is created for you in the specified path, which represents the application.
Application Package Prefix	`oracle.srdemo` This value becomes the prefix for all Java package names. You can override it later if necessary.
Application Template	`No Template [All Technologies]` In this tutorial, you access all of JDeveloper's technologies. New templates can be created and added to restrict the technologies that are available during development.

3. Click **OK**.

4. The Create Project dialog box appears. Set the values as follows:

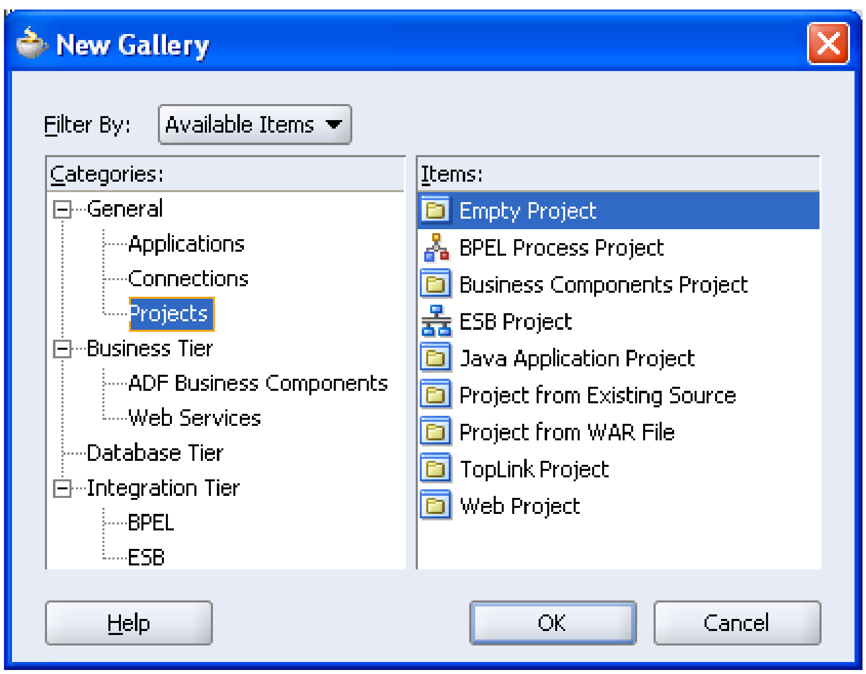
Field	Value
Project Name	`DataModel`
Directory name	<jdev_install>`\jdev\mywork\SRDemo\DataModel` Keep the default value. If you used the recommended directory structure, then your path should match this value. The directory is created for you in the specified path to contain the project's files.

5. Click **OK**.
The DataModel project represents the data model tier of your application. You build the components of this project in the next chapter by using Oracle TopLink.

Create another Project at the same level as the DataModel project.

6. In the Applications Navigator, right-click the **SRDemo** node and select **New Project**.
7. The New Gallery is invoked. Verify that **Empty Project** is selected. Then click **OK**.

   The new project becomes the child of a selected application node. To find out more about the types of projects and when they should be used, search Help for "Projects Category."



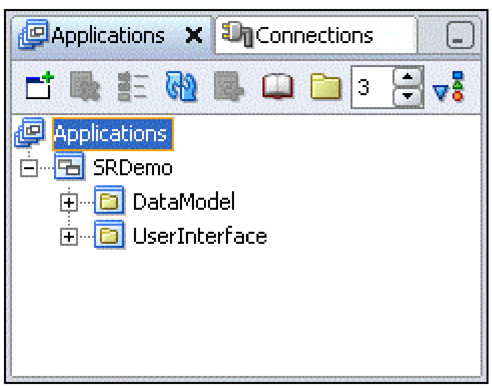
8. In the Create Project pane, set the values as follows.

Field	Value
Project Name	`UserInterface`

Directory name	<jdev_install>`\jdev\mywork\SRDemo\UserInterface` Keep the default value. If you used the recommended directory structure, your path should match this string.

9. Click **OK**. The UserInterface project represents the remainder of the application and contains the files that you create for the user interface in later chapters.
10. Double-click the new **UserInterface** project and select the **Dependencies** node.
11. Select the check box associated with the **DataModel** project. This enables the UserInterface project to access objects created in the DataModel project.
12. Select the **Project Content** node. At the bottom of the panel, set the Default Package to **`oracle.srdemo.userinterface`**. Then press **OK**. This enables you to better manage your classes and files.
13. Double-click the **DataModel** project and select the **Project Content** node.
14. At the bottom of the panel, set the Default Package to **`oracle.srdemo.datamodel`**. Then press **OK**. This enables you to better manage your classes and files.
15. Click the **Save All** button on the toolbar or select **File | Save All** from the menu.

Your Applications Navigator should look like the following screenshot. You are now ready to create application components for the tutorial.



Summary

In this chapter, you carried out all of the prerequisite steps that you need to complete before starting to build the application. You performed the following key tasks:

- Prepared the tutorial schema setup
- Installed the Service Request schema
- Started JDeveloper
- Created a JDeveloper database connection
- Defined an application and its projects in JDeveloper

2

Developing the Data Model

This chapter describes how to build the data model for your application using Oracle TopLink and Oracle ADF.

The chapter contains the following sections:

- Introduction
- Creating the Data Model Using Oracle TopLink
- Refining the TopLink Definitions
- Creating TopLink Named Queries
- Creating a TopLink Session
- Creating an EJB Session Bean
- Creating ADF Data Controls
- Summary

Introduction

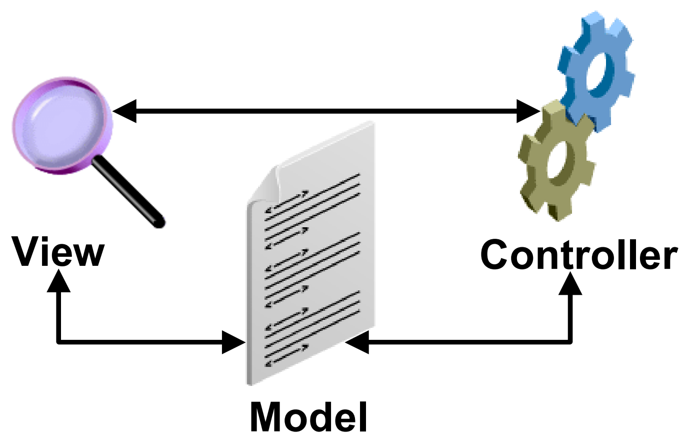
TopLink provides Java object-to-relational persistence, enabling you to create Java objects for accessing and persisting relational data. Oracle ADF lets you use these TopLink objects in your user interface through ADF data controls. These data controls enable client applications to use the data without concern for the underlying technology choice (in this case, Oracle TopLink).

You perform the following key tasks in this chapter:

1. Create TopLink POJOs (Java objects) for the database objects
2. Create a TopLink session
3. Create TopLink named queries
4. Create an EJB session bean
5. Create ADF data controls for the EJB session bean POJOs

The Data Model

As we mentioned earlier, good Java EE design consists of a clear separation of application function, namely the model, view, and controller. The **view** manages presentation of data to the user. The **controller** interprets user events (mouse and keyboard) from the user and dispatches the appropriate response. The **model** manages the data of the application. It also responds to changes in application state as directed by the controller.



The model contains specialized business rules that need to be applied to all data within the context of the application. It is also the correct object to govern default values required for any data elements.

The model should provide application specific data access as required. For example, if the application needs to query Employee data by Employee ID regularly, then the model should include a query element that meets this need.

Creating the Data Model Using Oracle TopLink

The application you are building in this tutorial uses Oracle TopLink for the Model layer.

TopLink is an Object-Relational Mapping layer that provides a map between the Java objects that the model uses and the database that is the source of their data.

In an MVC application, all data access performed by the application goes through the data model. This section shows you how to map the tables in your database to TopLink classes.

Creating TopLink Mappings for the Database Objects

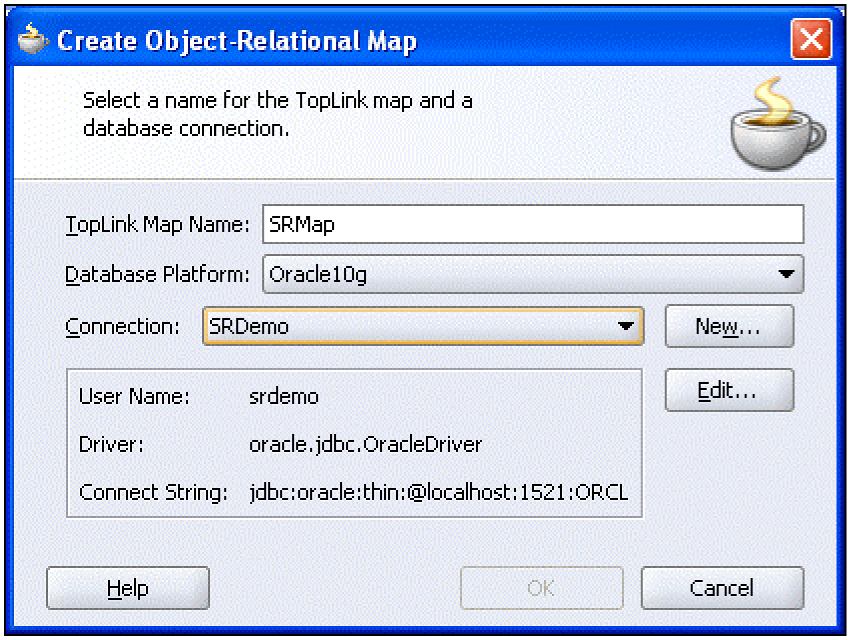
> **Note:** Underlying database schemas sometimes change after the TopLink mappings are created. In such cases, you can delete the objects and mappings and then re-create them. But it is better to have an agreed-upon database schema before creating the TopLink objects.

In this step, you reverse-engineer TopLink Java objects from existing database tables in the `SRDEMO` schema.

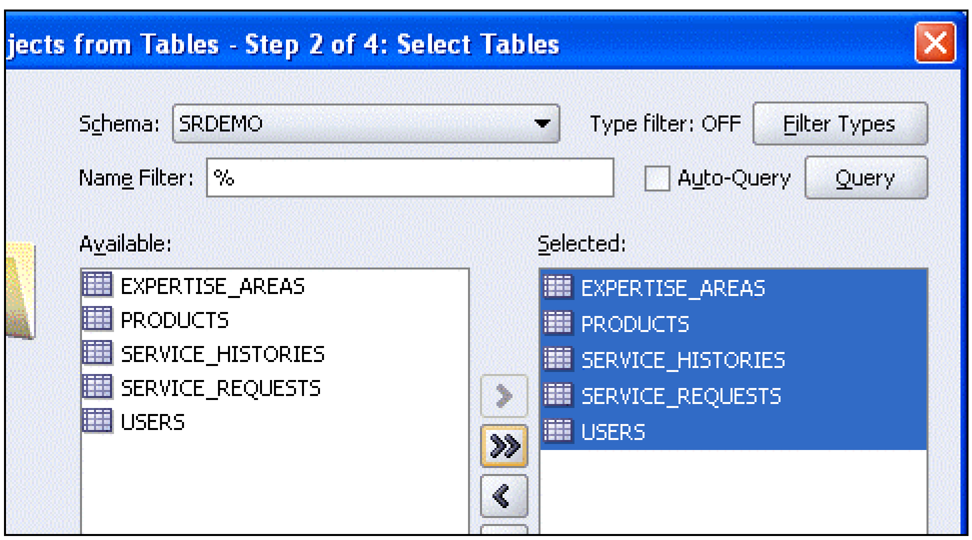
1. In the Applications Navigator, right-click the **DataModel** project and select **New**.
2. In the New Gallery, expand the **Business Tier** node, select **TopLink**, and choose **Java Objects from Tables** in the Items list.
3. Click **OK**.

TopLink requires a TopLink map file for each project. This file contains information about how classes are mapped to database tables. After the TopLink objects are created, you can edit this file to make changes to the default mappings. You need to create a new map for your project.

4. In step 1 of the “Create Java Objects from Tables“ Wizard, click the **New** button for the TopLink Map property.
5. In the Create Object-Relational Map dialog box, set the TopLink Map Name to **`SRMap`** and ensure that the connection is set to **`SRDemo`**.



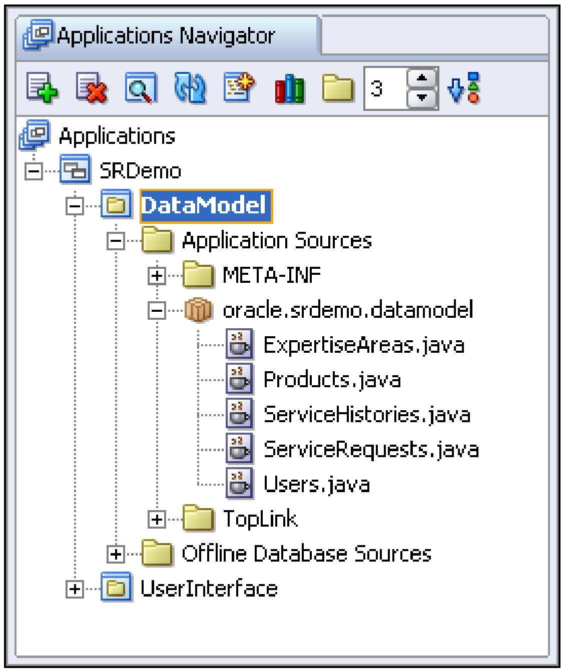
6. Click **OK**.
7. Return to step 1 of the wizard and click **Next**.
8. In step 2 of the wizard, select the tables that you want represented as TopLink objects. Click the **Query** button to display the available tables. Click the **>>** button to move all the available tables to the Selected side.



9. Click **Next**.
10. In step 3 of the wizard, confirm that the package name is `oracle.srdemo.datamodel`. If not, you need to modify it.
11. Click **Next**.

12. In step 4 of the wizard, notice the fully qualified Java class name for each of the database tables. Click **Finish** to complete the wizard.

13. Save your work by clicking the **Save All** button on the toolbar or by selecting **File > Save All** from the menu. Expand **Application Sources**. It should now look like this:



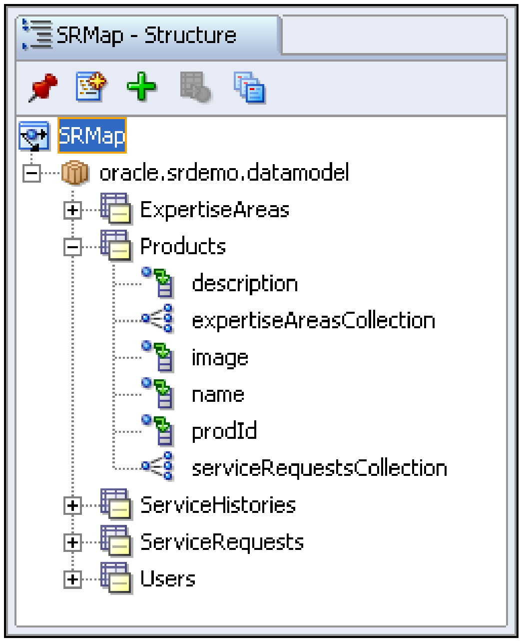
You have created TopLink POJOs for each of the five tables from the `SRDEMO` schema. Each of the `.java` files contains the code for the attribute definitions, constructors, getters, and setters.

14. Double-click the **`Products.java`** file to examine its contents. Notice that a variable is created for each of the columns from the table. Scroll down to view the constructors, getters, and setters.

The TopLink mapping file contains the object-to-table mappings. You can navigate to each object in the Structure window and view it in the editor.

15. In the Applications Navigator, click **SRMap** to select it. In the Structure window, expand the **oracle.srdemo.datamodel** node. Each node represents a created object.

16. Expand the **Products** node to reveal the mapped attributes and collections. It should look like the following screenshot:



Notice that there are several attributes and collections shown. Collections represent the relationship of the selected object to other objects in the model. For example, the `expertiseAreasCollection` collection represents the parent-child relationship to the `ExpertiseAreas` object.

Refining the TopLink Definitions

In this section, you refine the names of created TopLink objects, attach a sequence generator for primary key values, and specify some default values.

In some cases, you might want to define object names that are more meaningful than the default names assigned by TopLink. You can easily change an object's name.

The `ServiceHistories.java` file has both an integer for `svrId` and a `ValueHolderInterface` for service requests. The `svrId` integer is used in a non-object-oriented approach, so you delete that integer and just keep the collection for service requests. You also make similar changes to the `ExpertiseAreas.java` file.

1. In the Applications Navigator, double-click the **`ServiceHistories.java`** file, which opens the file in the code editor. In the editor, delete the line of code **`private Integer svrId`**. The variable `svrId` is used in setter and getter methods in this class. When you remove the variable, the code editor displays red bars in the right-hand gutter of the editor, where the variable was used but is not longer available. These are the methods you also need to remove.

2. Delete the getter (**`getSvrId`**) and setter (**`setSvrId`**) at the second and third location of these red bars (click the red bar to go to the particular method).

3. Repeat the previous two steps, this time for the **`ExpertiseAreas.java`** file. Delete **`private Integer prodId`** and **`private Integer userId`**, and then delete the getters and setters for those scalar attributes. There are six delete operations: two for the attributes,

two for the getters (`getProdId` and `getUserId`), and two for the setters (`setProdId` and `setUserId`).

4. Save your work.

Adding Code to Manage Default Values

If you need to provide for default values for attributes, you can add that code directly to the setter methods in the Java classes.

In the next few steps, you include default values for the `ServiceRequests` POJO:

1. Double-click **ServiceRequests.java** to open it in the code editor.
2. Find the `setRequestDate(Timestamp)` method and make the following changes. (This code sets the `requestDate` to use the current time and date as its default)

```
public void setRequestDate(Timestamp requestDate) {
    this.requestDate = (requestDate==null)
?new Timestamp(System.currentTimeMillis()):requestDate;
}
```

3. Change **`setStatus`** to use "**`Open`**" as its default.

```
public void setStatus(String status) {
   this.status = (status==null)?"Open":status;
}
```

Save your work and include more business logic to set default values and increment the line number for the `ServiceHistories` POJO. Make the following changes to the `ServiceHistories` POJO:

1. Set **`svhDate`** to use the current time and date as its default. Comment out the current assignment.

```
public void setSvhDate(Timestamp svhDate) {
  this.svhDate = (svhDate==null)
?new Timestamp(System.currentTimeMillis()):svhDate;
}
```

2. When the user adds a service history note, we want to determine the next line number and use it automatically. To determine the number, add a method that iterates through the current `ServiceHistories` collection to find the largest value, and then return that value incremented by 1. The method is used to calculate the next line number for a new service history record. It will iterate through the existing `ServiceHistories` to find the largest line number. Add the method as follows:

```
public Integer getNextLineItem() {
    int maxLineNo = 0;
    for (ServiceHistories
       svh:getServiceRequests().getServiceHistoriesCollection()){
        if (svh.getLineNo() !=null) {
            int testLineNo = svh.getLineNo().intValue();
            if (testLineNo > maxLineNo){
                maxLineNo = testLineNo;
            }
        }
    }
return ++maxLineNo;
}
```

3. Obtain the `setLineNo` value from the `getNextLineItem` method.

```
public void setLineNo(Integer lineNo) {
  this.lineNo = (lineNo==null)?getNextLineItem():lineNo;
}
```

4. In the Applications Navigator, select the **DataModel** project. From the context menu select **Rebuild**, to compile the classes. Fix any problems, then move on.

Using Toplink, you can declaratively use a database sequence to populate a column. In the next few steps, you select the primary key of the `ServiceRequests` table and set it to use a database sequence.

1. In the Applications Navigator, expand the **TopLink** node, and then select **SRMap**. In the Structure window, expand the **oracle.srdemo.datamodel** node under SRMap.

   The sequence generator supplies values for the `SERVICE_REQUESTS.SVR_ID` column, so attach it to the ServiceRequest mapping.

2. Double-click the **SRMap** and set the radio button to set the sequence to use **Native Sequence**.
3. Double-click the **ServiceRequests** node to see the mapping details for service requests. In the middle of the page, set the values for the Use Sequencing area of the page as follows.

Field	Value
Use Sequencing	Select the check box.
Name	`SERVICE_REQUESTS_SEQ`
Table	`SRDEMO.SERVICE_REQUESTS`

	(Should be the default)
Field	`SRDEMO.SERVICE_REQUESTS.SVR_ID` (Should be the default)

You have now finished the business logic changes for the data model. In the next section, you declaratively create some special methods for the pages.

3. Save your work.

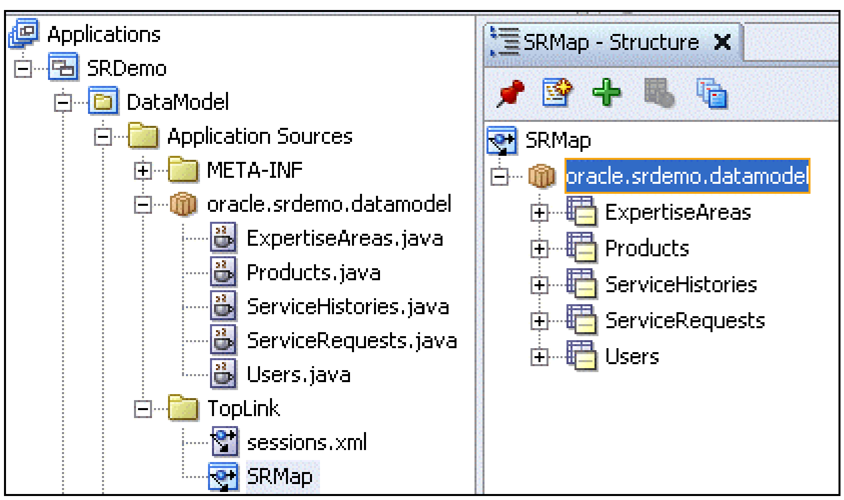
Creating TopLink Named Queries

Named queries provide a means of defining complex or commonly-used queries to access the database. Named queries offer some key advantages: they enable TopLink's internal performance optimizations to work better and make maintenance easier by centralizing the management of queries.

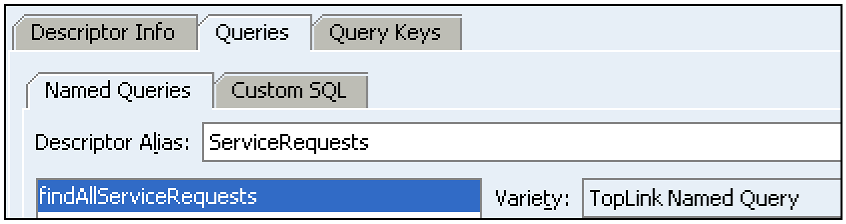
In the tutorial, you need to pass values between a Web client and TopLink with named queries. Each of these named queries includes a parameter and an expression that applies an incoming parameter to the query. You define both the query and the parameter expression in the next few steps.

Creating a Named Query

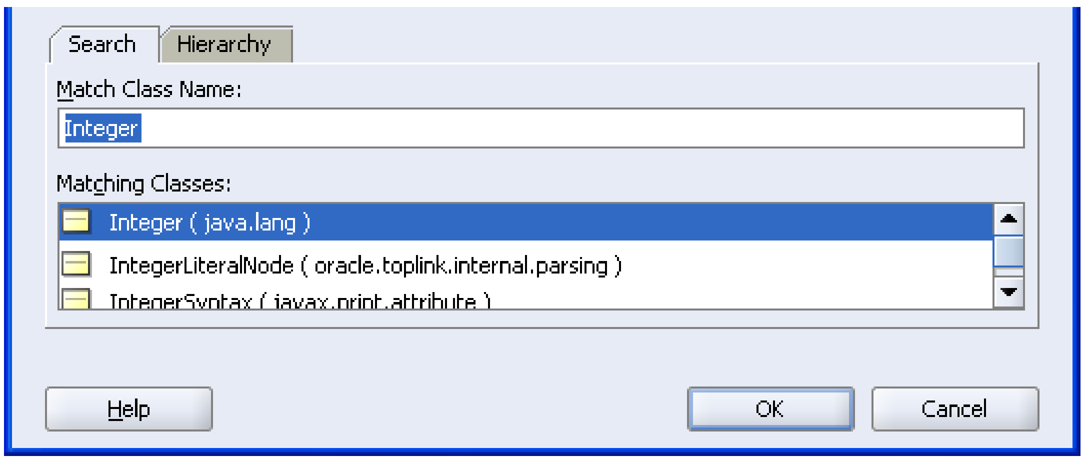
1. In the Applications Navigator, select the **SRMap** node under the TopLink node. In the Structure window, double-click the **SRMap** node and expand the oracle.srdemo.datamodel node. Each node represents a created object.



2. Click the **ServiceRequests** node, and then click the **Queries** tab. Click the **Add** button to create a new named query.



3. In the Add TopLink Named Query pane, type **`findServiceRequests`** as the name of the new query. Click **OK** to continue.
4. With the `findServiceRequests` named query selected, click the **General** tab. In the Parameters area of the editor, click the **Add** button.
5. In the Class Browser window, click the **Search** tab. This is where you define the parameter type.
6. You can simply type **`Integer`** and you will see **Integer (`java.lang`)**. When the `java.lang.Integer` matching class is highlighted, click **OK** to define the type.

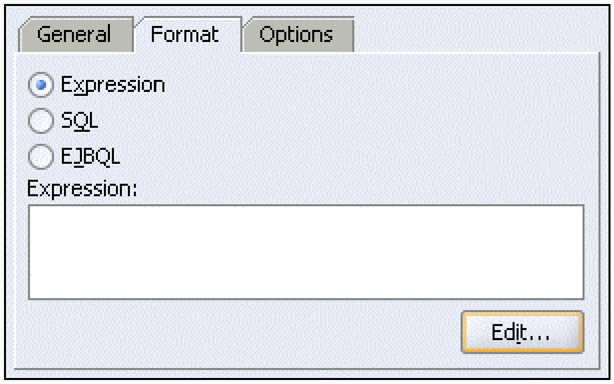


7. In the Parameters area, change the parameter name to **`filedBy`**.
8. Create a second parameter of `java.lang.String` and name it **`status`**.

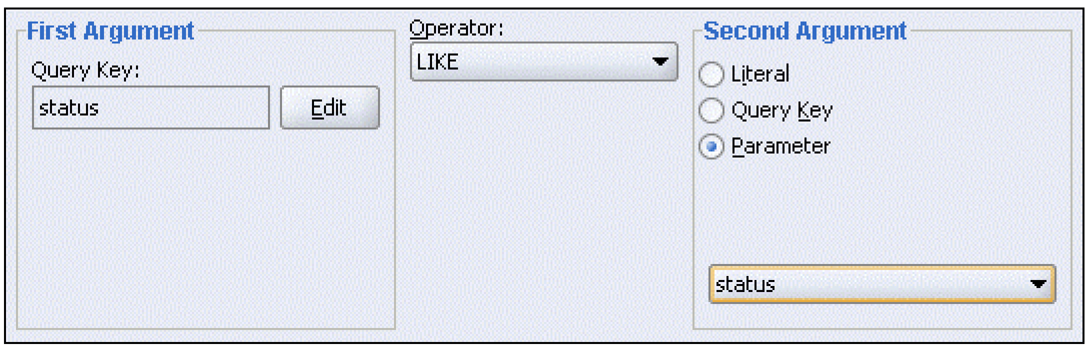
Defining a Query Expression

Now that you have defined two parameters (`filedBy` and `status`), you need to define an expression using them. This expression defines how the parameter is associated with an attribute in the named query. The expression is evaluated at run time to determine the rows returned by the named query.

1. With the findServiceRequests named query selected, click the **Format** tab and then click the **Edit** button in the Parameters area of the editor.



2. In the Expression Builder, click **Add** to create a new expression.
3. In the First Argument area of the expression, click **Edit**. In the Choose Query Key pane, select **`status`**.
4. Click **OK** to continue.
5. Back in the Expression Builder, set the Operator to **`Like`**, and the Second Argument to the `status` parameter that you created earlier. It should look like the following screenshot.



6. When the first component of the expression is complete, select it and click the **Add Nested** button.
7. Select the **2.AND** node and change the Logical Operator to **OR**.
8. Set the nested expression component to the values in the following table. Select the check box to **Allow Nulls** in the expression.

Field	First Argument	Operator	Second Argument (Parameter)
Second Component	`AssignedTo(AllowsNull).userId`	`EQUAL`	`filedBy`

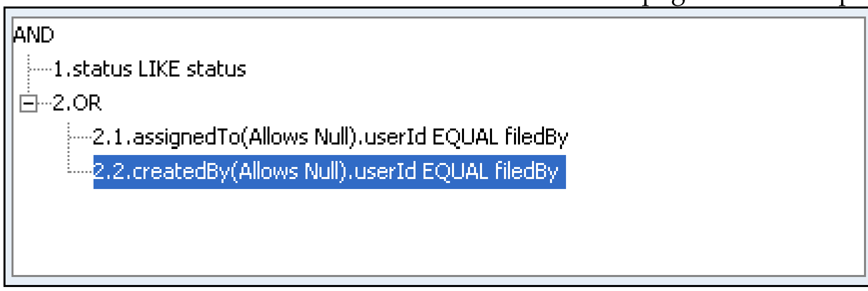
9. Create a second nested expression component at the same level as the previous one. Click the **Add** button to do this (*not* the Add Nested button). Set the values to those in the following table. Select the check box to **Allow Nulls** in the expression.

Field	First Argument	Operator	Second Argument

			(parameter)
Third Component	CreatedBy(AllowsNulls).userId	EQUAL	filedBy

When complete, your expression should look like the following. Confirm your results, make changes if necessary, and click **OK** when done.

Make sure you have changed the logical operator to **OR**. If the logical operator is **AND**, both conditions must be true and therefore the SRList page will not display any rows.



Creating More Named Queries

1. Create another named query on the ServiceRequests POJO using the values provided in the following table: Set the **Type** to **ReadObjectQuery**.

Field	**Value**
Name	`findServiceRequestById`
Parameter Type	`java.lang.Integer`
Parameter Name	`findSvrId`

2. Create an expression for the named query using the values in the following table:

Field	**Value**
First Argument	`svrId`
Operator	`EQUAL`
Second Argument	`findSvrId`

3. Now create some named queries in other POJOs. Create a named query on Users using the values provided in the following table: Set the **Type** to **ReadObjectQuery**.

Field	**Value**
Name	`findUserById`

Parameter Type	`java.lang.Integer`
Parameter Name	`createdBy`

4. Add an expression for the named query using the values in the following table:

Field	Value
First Argument	`userId`
Operator	`EQUAL`
Second Argument	`createdBy`

5. Finally, create a named query on Products using the values provided in the following table: Set the **Type** to **ReadObjectQuery**.

Field	Value
Name	`findProductById`
Class Name	`java.lang.Integer`
Parameter Name	`prodId`

6. Create an expression for the named query using the values in the following table:

Field	Value
First Argument	`prodId`
Operator	`EQUAL`
Second Argument	`prodId`

7. Save all your work. To ensure that you've done all the steps correctly, right-click the **SRMap** and select **Generate Mapping Status Report** from the context menu.

   In the messages area, the Mapping Status–Log should show that the SRMap does not contain any errors. If yours does contain errors, recheck the values you entered for the steps in this section.

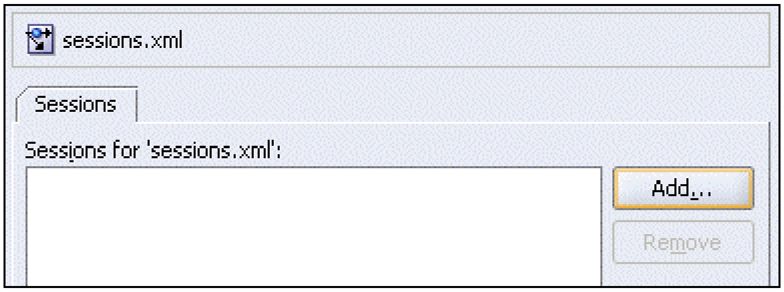
Creating a TopLink Session

You now create a TopLink session. To do this, you first define a session configuration file. The TopLink session configuration is used at run time to maintain state information about each running session. It also supplies database connection information for the session. Each active connection to the application receives a unique session object.

By default, a session is created named **`default`**. In the following steps, you create a new session

with a name that you define:

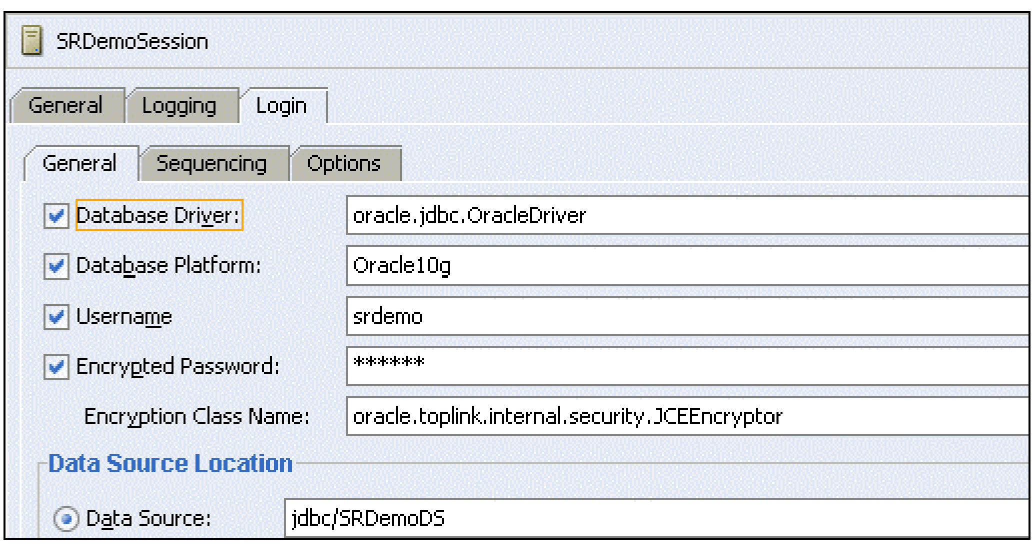
1. In the Applications Navigator, select the **`sessions.xml`** file. In the Structure window, select the **`default`** session and delete it.
2. Double-click the **`sessions.xml`** file in the Structure window and click the **Add** button in the editor.



3. Name the new session **`SRDemoSession`**, and then click **OK**. Leave the remaining properties set to their defaults.
4. Click **Save All** to save your work.

In the next few steps, you enable logging so that you can see the detailed results of running your application. This is useful if you need to debug. You also set the data source location to use a data source definition instead of a direct database connection. This will prove useful during the deployment phase of this application development. You can then change the connection details without affecting any code or settings in the application.

5. In the Structure window, select the **`SRDemoSession`** session. The editor displays the session name and three tabs: General, Logging, and Login.
6. Click the **Logging** tab and set the Enable Logging property to **`True`**.
7. On the Login tab, select the **Data Source** option and set its value to **`jdbc/SRDemoDS`**. This value is case sensitive, so make sure the case matches the case that you used for your connection name.



8. Click **Save All** to save your work.

In addition to specifying connection information, you can also switch logging features either on or off for the session. The logging feature records session information at a level you choose. For example, you can choose to log debugging information, application exceptions, or both.

In the next steps, you create a session bean named `SRPublicFacade` in the `oracle.srdemo.datamodel` package.

Creating an EJB Session Bean

In this section, you create an EJB session bean to provide an access point for your application. The client application uses this session bean for its access to the data model. This approach follows the Session Facade design pattern for constructing applications.

Session beans encapsulate business logic and business data while exposing the necessary interfaces. As such, the client tier can make use of the distributed services within the model without concern for its complexity. The session bean is made up of two files:

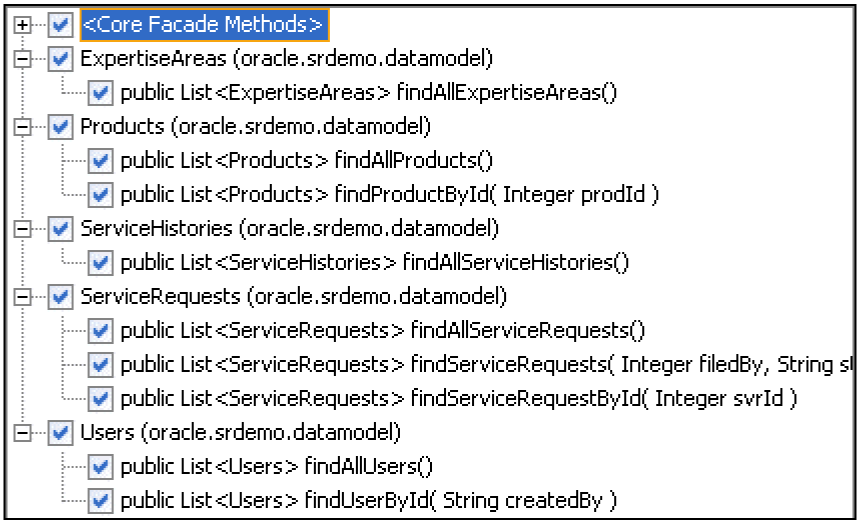
- **`SRPublicFacadeBean`** contains the bean code.
- **`SRPublicFacadeLocal`** is a local interface for the session bean.

In these next few steps you create a Session Bean and interface to represent the TopLink POJOs.

1. Create a session bean named **`SRPublicFacade`** in the `oracle.srdemo.datamodel` package**.** In the Applications Navigator, select the **DataModel** project and then select **New** from the context menu.
2. In the New Gallery, expand the **Business Tier** and click the **EJB** node. In the Items pane, select the **Session Bean** and then click **OK**.
3. In the Create Session Bean Wizard, "Step 1: EJB Name and Options," set the values as follows and then click **Next**:

Field	Value
EJB Name	`SRPublicFacade`
Session Type	`Stateless`
Transaction Type	`Container`
Generate Session Façade Methods	Select the check box.
Entity Implementation	Select the TopLink POJOs option.

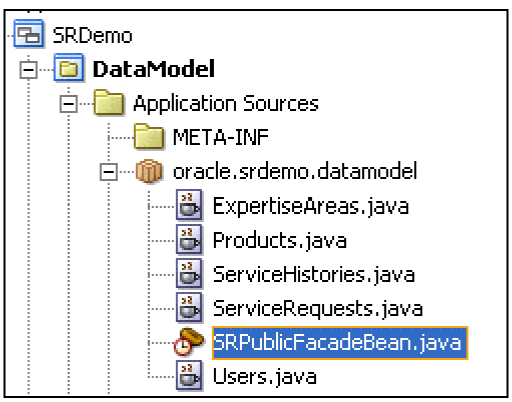
In "Step 2: Session Facade," you want the Session Facade pattern to incorporate all the methods and named queries. Expand the **`ServiceRequests`** node to expose your declaratively created named query. Ensure that all check boxes are selected, and then click **Next**.



4. In “Step 3: Class Definition,” ensure the Bean Class is set to **`oracle.srdemo.datamodel.SRPublicFacadeBean`**. The default directory is acceptable. Click **Next**.

5. In the final wizard step, clear the **Implement a Remote Interface** check box and leave **Implement a Local Interface** check box selected. You need this selection when you are working with a Web client, as you are here.

6. Click **Finish** to complete the process. The Applications Navigator now displays the new `SRPublicFacadeBean` class (as shown in the following screenshot):



Customizing the Session Bean

The application you are building focuses on creating and maintaining service requests. The best practice for creating new rows is to add a method to the session bean that accepts parameters and creates a new row based on those parameters. Because this is a method in the session bean, you can add whatever code is required, no matter how simple or complex. For your convenience, the code for the create service request method is provided in the setup files for this tutorial.

In the next few steps, you copy this code into the session bean.

1. Select **`SRPublicFacadeBean`**, and expand the **`Sources`** node in the Structure window. Double-click the **`SRPublicFacadeLocal`** node to open the file in the editor.
2. The code you need to enter can be found in a file from the <tutorial_setup>\files\ directory. In Windows Explorer, find and open the **`CreateInterface.txt`** file. Copy the code and paste it at the bottom of the `SRPublicFacadeLocal` file (before the closing brace).
3. Add the detail methods in `SRPublicFacadeBean`. Double-click **`SRPublicFacadeBean`** to invoke it in the editor. In the <tutorial_setup>\files\ directory, open the **`CreateSession.txt`** file. Copy all the code and paste it into `SRPublicFacadeBean.java`.
4. Save your work.

Creating ADF Data Controls

To make use of your TopLink POJOs and `SRPublicFacadeBean`, you need to create ADF data controls.

Data controls provide the interface that binds the data model to your user client application. This means that when you later develop your client application, you can base your page components on the data controls without concern for the actual technology that was used to build the data model (in this case, TopLink and EJB session beans).

You now create a data control that provides access to the objects defined in your TopLink map (`SRMap`), as follows:

In the Applications Navigator, right-click the **`SRPublicFacadeBean.java`** node and choose **Create Data Control** from the context menu.

The details of the data control are maintained in several files. These include XML files that determine which built-in operations are allowed for each object. Here is a summary of the files created for the data control.

File Name (or Example)	File Type
`Users.xml, ServiceRequests.xml, etc.`	**Represents an individual POJO (one for each) and describes its attributes, accessors, and parameters**
`UpdateableSingleValue.xml`	**Controls operations on objects with single values**
`UpdateableCollection.xml`	**Controls operations on objects with multiple values**
`DataControls.dcx` (in `oracle.srdemo` package)	**Defines the interface used for the data control**

You can also add properties to a bean that will alter the run-time behavior of the data control. Some of the most common properties are:

- **Control hints:** Provide label text, tool-tip text, and formatting rules
- **Attribute rules:** Enable primary key definition and query rules

- **Declarative validation rules** (such as value comparison, list validation, range validation, and expression validation)

The Bean Properties Editor gives you a tool to customize bean behavior without requiring Java code in the bean class. The changes you make using this editor are stored in the XML file for each POJO (such as `Users.xml`).

You can access the bean properties editor from the Structure window:

1. Click the **`Users.xml`** file in the Applications Navigator.
2. Right-click **Users** (the top-level object) in the Structure window and select **properties** from the context menu to open the Bean Properties Editor. You can now explore the multiple options for defining declarative rules for a specific JavaBean.
3. Click Cancel when you have finished exploring the options.

Build Your Application

As the last step in this chapter, you compile your application to ensure that everything you did worked.

1. Right-click **SRDemo** and select **Rebuild** from the context menu.
2. Check the log window to make sure that there are no compile errors (or other errors).

Note: As part of the tutorial setup files, we have included a zipped application for each chapter with the steps completed successfully.

If your application does not compile correctly or if it has other errors, you can use these applications as a starting point for the following chapter. There are instructions on how to use these starter applications at the beginning of each chapter.

Summary

In this chapter, you built the data model for your application. To accomplish this, you performed the following key tasks:

- Created TopLink POJOs (Java objects) for the database objects
- Refined the TopLink definitions
- Created a TopLink session
- Created an EJB session bean
- Generated ADF data controls for each of the objects defined in your TopLink map
- Created TopLink named queries for the data model

3

Defining Page Flow and Navigation

This chapter describes how to create outline definitions for each of the pages in your JSF application and specify the navigation between them. You do this using a diagrammer in JDeveloper.

The chapter contains the following sections:

- Introduction
- Creating a JSF Navigation Model
- Creating Page Icons on the Diagram
- Linking the Pages Together
- Summary

Introduction

In the previous chapter, you built the data model for the Service Request application. Ensure that you have successfully built the data model, as described in that chapter, before you start to work on this one.

In this chapter, you start to work on the user interface. You use the JSF Navigation Modeler in JDeveloper to diagrammatically plan and create your application's pages and the navigation between them. You perform the following key tasks:

- Creating a page-flow diagram
- Creating page placeholders on the diagram
- Defining the navigation rules and navigation cases between the pages

---

Note: If you did not successfully complete Chapter 2, you can use the end-of-chapter application that is part of the tutorial setup:

1. Create a subdirectory named **`Chapter3`** to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter3`.
2. Unzip **`<tutorial-setup>\starterApplications\SRDemo-EndOfChapter2.zip`** into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter3\SRDemo\SRDemo.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 2.

Creating a JSF Navigation Model

With the JSF Navigation Modeler, you can visually design your application from a bird's-eye view. In the following steps, you open a new empty diagram and add a title to it:

1. In the Applications Navigator, right-click the UserInterface node, and select **New** from the short cut menu.
2. In the New Gallery, expand the **Web Tier** node and select **JSF** in the Categories pane.
3. Choose **JSF Page Flow & Configuration (faces-config.xml)** from the Items pane. Then click OK.

   This choice adds a JSF configuration file to your project. You edit the JSF configuration file by using the JSF Navigation Modeler to specify navigation rules between the pages of your application.

4. In the Create JSF Configuration File dialog box, accept the default name **`faces-config.xml`** and create it in the default directory location. Click **OK** to continue.

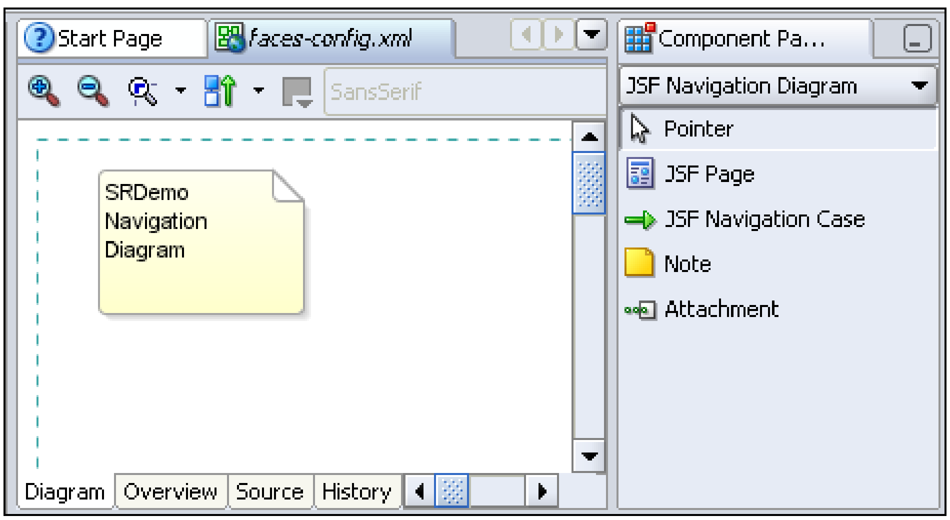
   The empty diagram opens. Notice the Component Palette to the right of the diagram editor. You use this to create components for the JSF Navigation Model.

   Notice also the four tabs at the bottom of the diagram editor screen. The default view is the Diagram view, where you can model and create the pages in your application. Clicking the Overview tab shows a console-type interface that enables you to register any and all types of configurations into your `faces-config` file, including managed beans, navigation rules, and other items such as custom validators, converters, and so on. The Source tab enables you to edit the generated XML code directly. JDeveloper automatically synchronizes the different views of your JSF navigation. Finally, the History tab shows a history of recent changes.

5. Add a title to your diagram. Select **Note** from the Component Palette, where the JSF Navigation Diagram choices are listed, and then drag it to the upper-left portion of your diagram.

   It is good practice to provide annotation to a diagram, and the general Note component is useful for this purpose.

6. Type **`SRDemo Navigation Diagram`** in the box on the diagram, and then click elsewhere in the diagram to create the note. The note serves as your diagram's title.



Note: The steps in this section showed you how to explicitly create a new page flow diagram. You need to do this because you are just starting to build the Web application and have not yet created any pages. If you don't create a `faces-config.xml` file explicitly, JDeveloper creates one automatically when you add a JSF page to your project. Then you simply need to double-click it in the Navigator to open it in the diagram editor.

Creating Pages on the Diagram

There are two ways to add a page to a page flow diagram. If you have already created some JSF pages, you can drag them from the Navigator to the diagram. Alternatively, you can create the pages directly in the diagram.

You have not yet created any pages for the SRDemo application. In the following steps, you create placeholders for pages in the application. (You define the pages fully in subsequent chapters of the tutorial.)

1. Select **JSF Page** in the Component Palette, and click where you want the page to appear in the diagram.

   An icon for the page is displayed on the diagram. At this stage, the icon initially has a yellow warning over it to remind you that it is simply a placeholder rather than a fully defined page.

2. Click the icon label and type **`/app/SRList.jspx`** as the page name.

   The SRList page is at the center of the application. It is a page where all users (customers, technicians, and managers) can browse existing service requests.

   The page name requires an initial slash so that it can be run. If you remove the slash when you type the name, it is reinstated. You can create the detailed pages either iteratively (as you develop the page flow diagram) or at a later stage.

   For the purposes of the tutorial, you create placeholders for all the pages in the application at this point and build the detailed pages in later chapters.

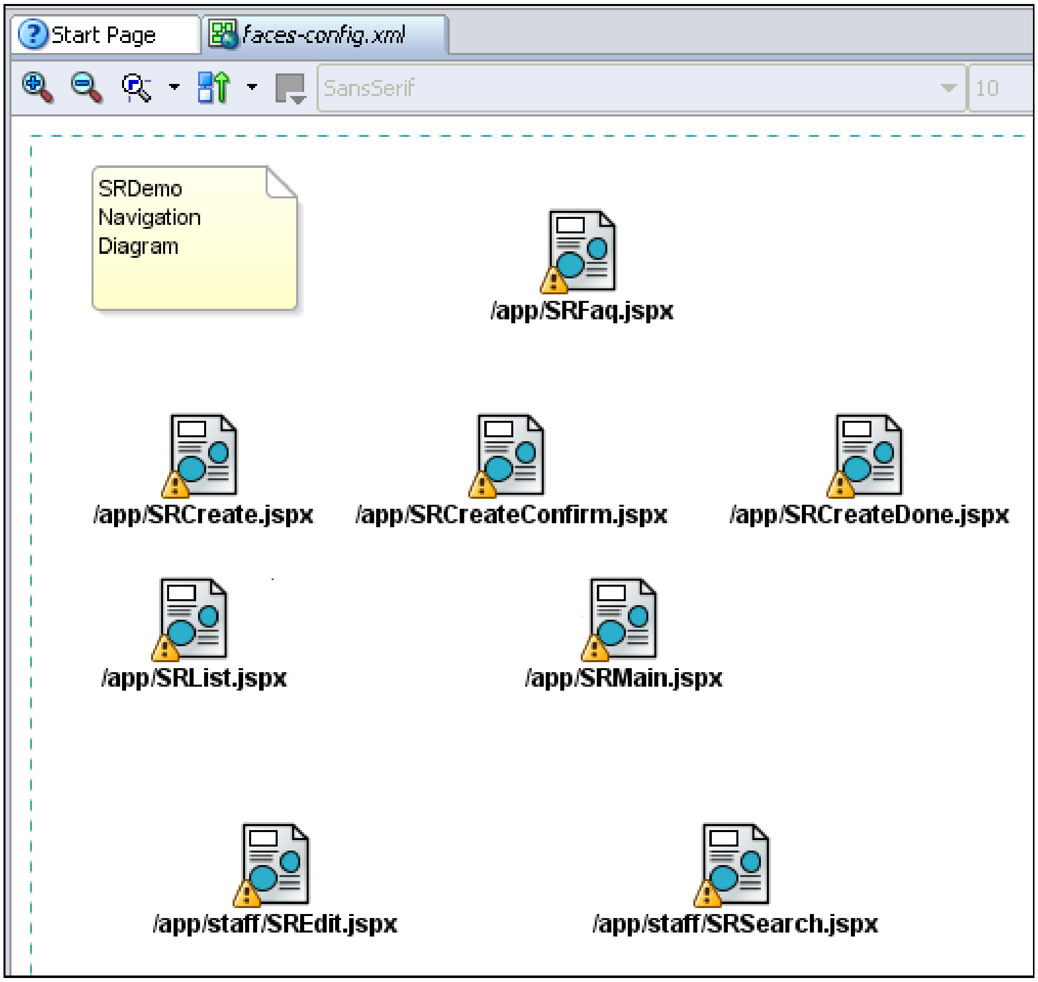
3. Repeat steps 1 and 2 to create five more page placeholders in the `/app` directory:

Page Name	Page Purpose
`/app/SRMain.jspx`	Enables users to add extra information to an existing service request
`/app/SRCreate.jspx`	Enables users to create a new service request
`/app/SRCreateConfirm.jspx`	Confirmation screen for new requests
`/app/SRCreateDone.jspx`	Final screen in the creation of a new request
`/app/SRFaq.jspx`	Enables users to view commonly asked questions

4. Then create the following two staff-specific pages in the `/app/staff` directory:

Page Name	Page Purpose
`/app/staff/SREdit.jspx`	Enables managers and technicians to amend service requests
`/app/staff/SRSearch.jspx`	Enables users to search for a service request

These screens make up the SRDemo application. When you have finished, your diagram should look like the following screenshot. If it does not, you can drag the pages to make your diagram similar.



Linking the Pages Together

Now that you have created placeholders for the application’s pages, you need to define how users navigate between them. JSF navigation is defined by a set of rules for choosing the next page to be displayed when a user clicks a UI component (for example, a command button). These rules are defined in the JSF configuration file, which is created as you create the diagram. There may be several ways in which a user can navigate from one page, and each of these is represented by a different navigation case.

For the SRDemo application, you start by drawing simple navigation cases on the page flow diagram.

Add a link to enable a user to navigate from the SRList page to the SRMain page:

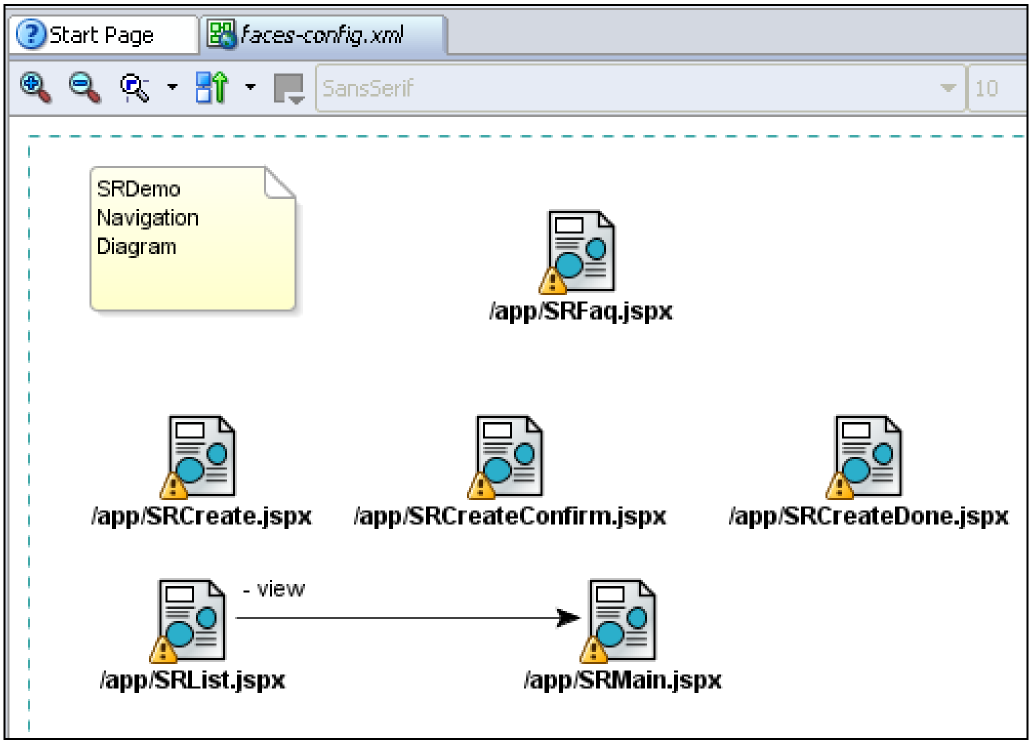
1. Select **JSF Navigation Case** in the Component Palette.
2. Click the icon for the source JSF page (**SRList**), and then click the icon for the destination JSF page (**SRMain**) for the navigation case.

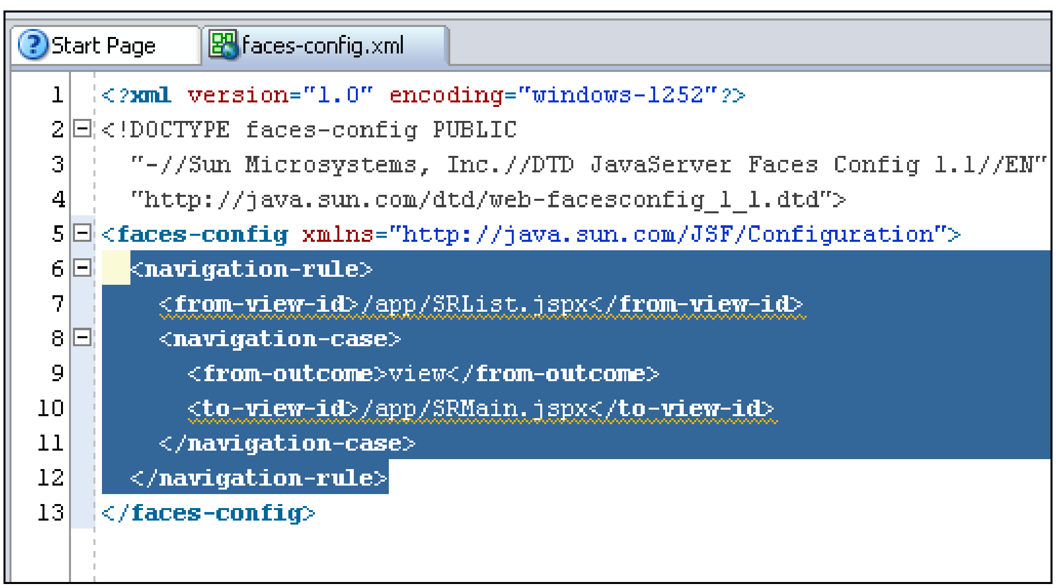
The navigation case is shown as a solid line on the diagram, and a default label ("success") is shown as the name of the navigation case.

3. Modify the default label by clicking it and typing **view** over it. The SRList page has a View button, which users click to navigate to the SRMain page.

4. Click the **Overview** tab at the bottom of the screen. Click **Navigation Rules** in the left table. Notice that the rule you just created in the diagram is listed in the table.

5. To understand the syntax of the navigation rule you created, click the **Source** tab to see the XML code for the rule. The following images show you the rule on the diagram and in the source:





The `<from-view-id>` tag identifies the source page; the `<to-view-id>` tag identifies the destination page. The wavy lines under the page names remind you that the pages have not yet been created.

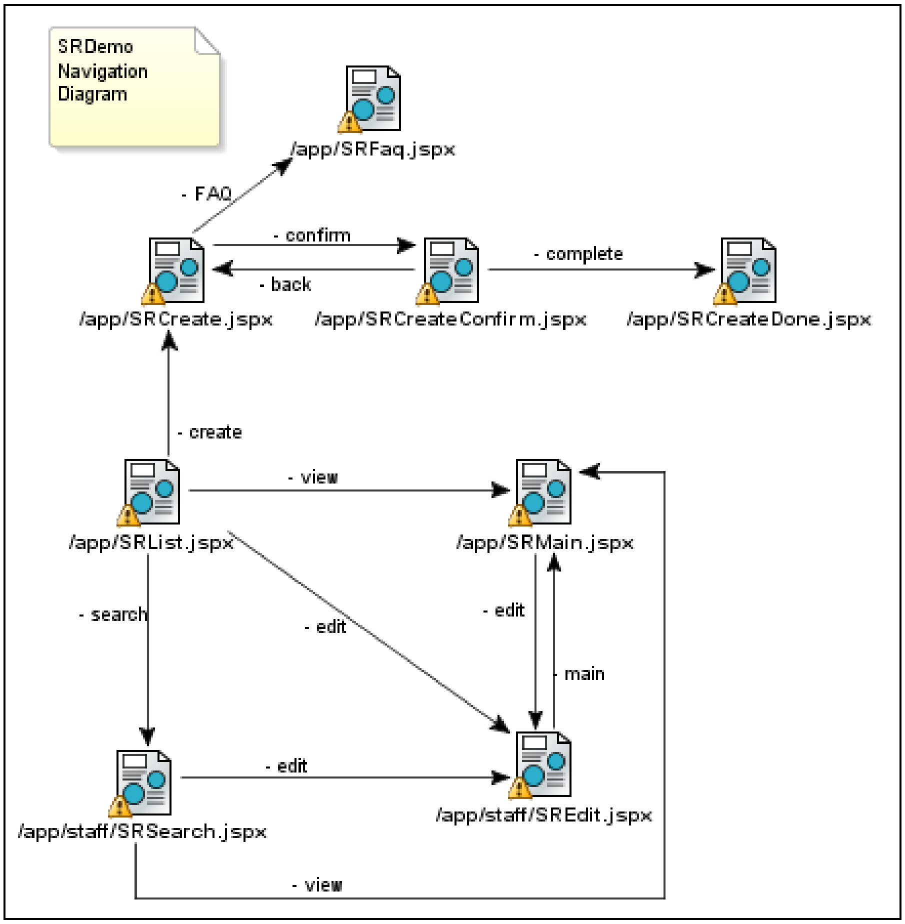
6. Repeat the steps to create more navigation cases on your diagram, as defined in the following table.

   The table shows the paths by which users navigate between the pages shown. For example, a user clicks a View button on the SRList page to navigate to the SRMain page, and clicks an Edit button to navigate to the SREdit page.

Source	Destination	Outcome
SRList	SRMain	view
SRList	SREdit	edit
SRList	SRSearch	search
SRMain	SREdit	edit
SREdit	SRMain	main
SRSearch	SRMain	view
SRSearch	SREdit	edit
SRList	SRCreate	create
SRCreate	SRFaq	FAQ
SRCreate	SRCreateConfirm	confirm
SRCreateConfirm	SRCreate	back
SRCreateConfirm	SRCreateDone	complete

Note: To create a "dogleg" in the line representing the navigation case, click once at the point where you want the change of direction to occur. You can add any number of doglegs to a navigation case.

7. Click **Save** to save the diagram. Your diagram should look like the following screenshot:



Defining Global Navigation Rules

Global navigation rules define the navigation paths that are available from all pages (typically through the use of Help buttons and Logout icons). You don't use the diagram to define these paths; you use the Overview page (or Configuration Editor) instead.

To define global navigation rules, perform these steps:

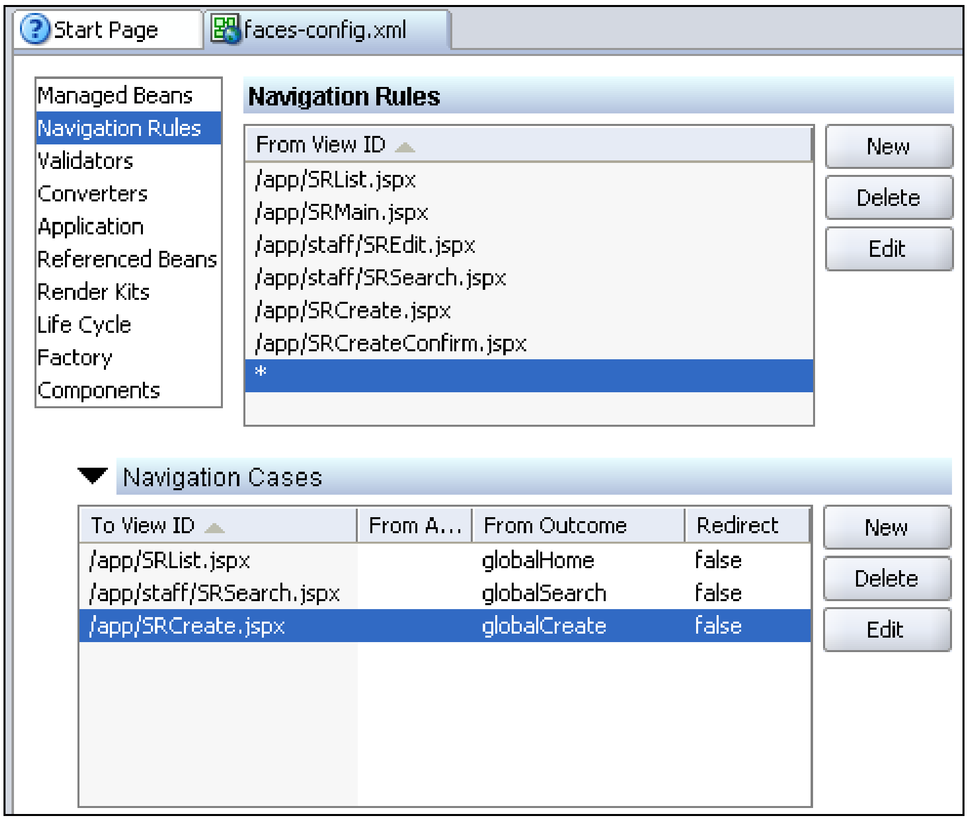
1. Click the **Overview** tab at the bottom of the diagram page.
2. Select **Navigation Rules** on the top left of the page. The Navigation Rules box displays the rules you just created in the diagram.
3. Click the **New** button on the right.
4. In the Create Navigation Rule dialog box, type the wildcard symbol *, and then click **OK**.
5. Click the **New** button to the right of the Navigation Cases box.
6. Type **`/app/SRList.jspx`** in the To View ID field (the drop-down list is empty because you have not yet created any pages). Type **`globalHome`** in the From Outcome field. Then click **OK**. This identifies the SRList page as the home page , to which users can return from any page in the application.

7. Create two more global rules as defined in the following table:

To View ID	From Outcome
`/app/staff/SRSearch.jspx`	`globalSearch`
`/app/SRCreate.jspx`	`globalCreate`

8. Save your work

The completed Navigation Rules page should look like the following screenshot:



Summary

In this chapter, you created a page-flow diagram showing the pages of your application and the links needed for users to navigate between them. To accomplish this, you performed the following key tasks:

- Created a new page-flow diagram
- Created outline pages on the diagram
- Created navigation links between the pages
- Defined global navigation rules

4

Developing Application Standards

This chapter discusses the role of standards in application development and demonstrates how to implement them in the SRDemo application.

The chapter contains the following sections:

- Introduction
- Reusing Code
- Providing for Translation of the User Interface
- Providing a State Holder
- Creating a Standard Look and Feel
- Summary

Introduction

There are different sets of standards that need to be defined within each individual project, no matter how small the project is. This includes ensuring a common look and feel across the user interface, avoiding duplication of code where possible, and facilitating translation to other languages. This chapter considers these types of standards in the context of the tutorial.

You perform the following key tasks:

- Creating utilities files to facilitate the reuse of code across the application
- Providing for translating the user interfaces into multiple languages
- Creating a basic template page to provide a standard look and feel for the application

---

Note: If you did not successfully complete Chapter 3, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named **`Chapter4`** to hold the starter application. If you used the default settings, it should be in **`<jdev-install>\jdev\mywork\Chapter4`**.
2. Unzip **`<tutorial-setup>\starterApplications\SRDemo-EndOfChapter3.zip`** into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter4\SRDemo\SRDemo.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 3.

Reusing Code

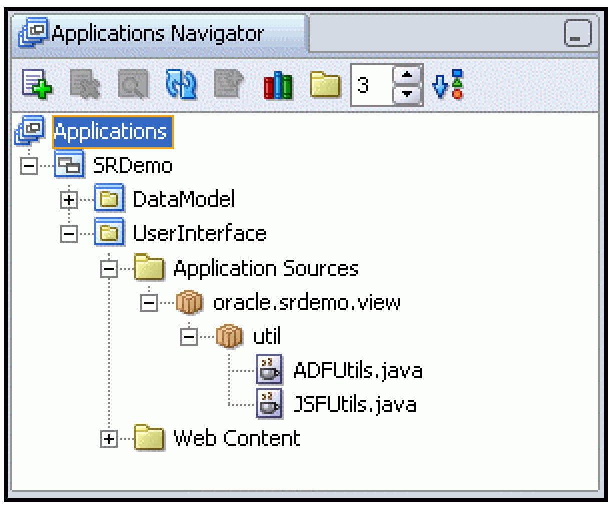
It makes sense, of course, to write code once and reuse it. To aid reuse in SRDemo, the `ADFUtils.java` and `JSFUtils.java` files have been created to hold useful utilities that you may need at points throughout the application. Both these files are provided with the tutorial.

Perform the following steps to create Java files in your application to hold this code:

1. In the Navigator, select the **UserInterface** project and then choose **New** from the context menu.
2. In the New Gallery, select the **General** node in the Categories pane (if it is not already selected), and in the Items pane choose **Java Class**. Click **OK.**
3. Type **`ADFUtils`** as the class name, and specify **`oracle.srdemo.view.util`** as the package name. Ensure that the Extends field is set to **`java.lang.Object`** and leave the other fields at their defaults. Click **OK**. The Code Editor opens and displays the code for the skeleton class.

4. In Windows Explorer (or the equivalent if you are using another operating system), navigate to the directory where you unzipped the tutorial setup files. Then open the **`ADFUtils.java`** file (which you should find in the `<ADFTutorialSetup>\files` directory).
5. Select all the text in the file and copy it.
6. In JDeveloper, delete the skeleton code from the `ADFUtils.java` file, and then paste the contents of the clipboard into the file in its place. Save the file.
7. Examine the contents of the file. It holds a series of convenience functions for dealing with ADF Bindings. Note that the red wavy lines under some lines of code in the file indicate that the ADF Faces Runtime library is missing from the project. You add this library later in the chapter.
8. Repeat these steps to create a second utilities file, **`JSFUtils.java`**. This file contains some general static utilities.

The following screenshot shows how the two files should appear in the Navigator.



Providing for Translation of the User Interface

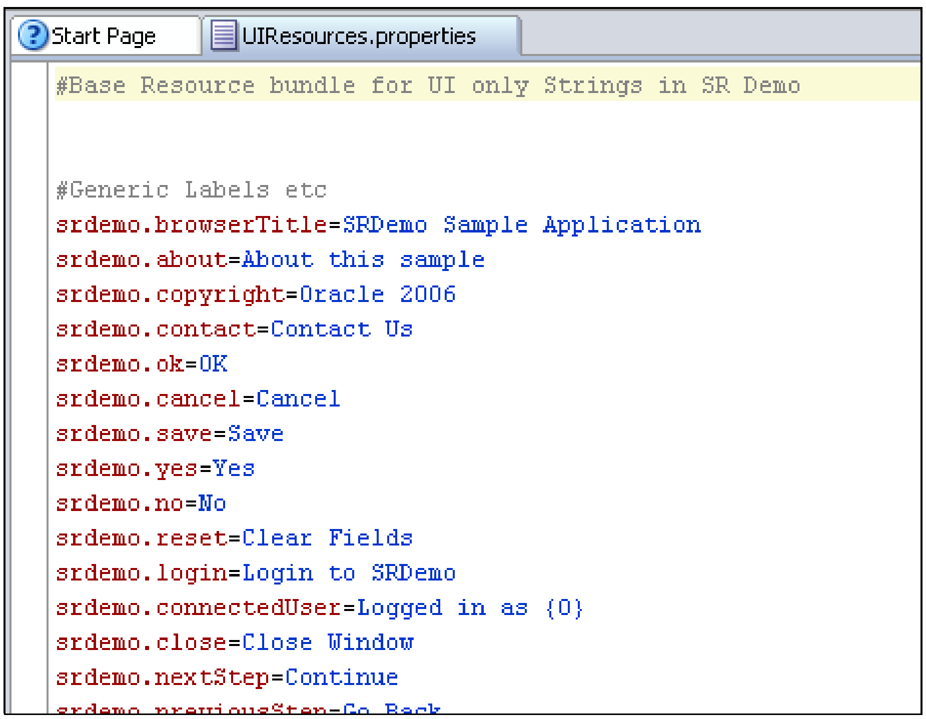
JavaServer Faces (JSF) makes the process of translating the user interface into multiple languages relatively straightforward. Most of the strings used in the user interface of SRDemo are defined in a single file, `UIResources.properties`. This is a flat text file containing name=value pairs, where the name is an abstract identifier of a resource and the value is the actual value that is displayed at run time. For example, `srdemo.browserTitle`=SRDemo Sample Application.

This kind of properties file makes maintenance of the UI simpler and also helps greatly with translation. To translate the application, all you have to do is duplicate the properties file by selecting File > Save As and saving with a file name with the appropriate suffix (for example, `UIResources_de` for the German version). You then translate the values (leaving the names of the resources the same). JSF inspects the browser locale information and automatically loads the correct bundle at run time. If a resource is not found in the locale-specific bundle, JSF uses the base bundle instead.

For convenience, the `UIResources` file is supplied with the tutorial. Perform the following steps to import the file into your JDeveloper project:

1. In the Navigator, select the **UserInterface** project, and then choose **New** from the context menu.
2. In the New Gallery, select the **General** node in the Categories pane (if it is not already selected), and then choose **File** in the Items pane. Click **OK.**
3. Name the file **`UIResources.properties`** and click **OK**.

   JDeveloper creates a Resources node under the `UserInterface` project and places the file there.
4. In Windows Explorer (or the equivalent in your operating system), navigate to the directory where you unzipped the setup files. Open the **`UIResources.properties`** file (it should be in the `<tutorial_setup>\files` directory), select all the text in the file, and copy it to the clipboard.
5. In JDeveloper, paste the contents of the clipboard into the file and save it.
6. Examine some of the name=value pairs described above. You use them throughout your pages. The following screenshot shows some examples:



Providing a State Holder

As you navigate between pages in the application, you need to track some values. These values represent things like the current service request ID as you move from the SRList page to either the SRMain or SREdit pages. You need to track which user the queries records are for on the SRSearch page as you get ready to edit them.

In this section, you create a class to manage all the attributes needed to track the state of the

application. Then you create a managed bean to expose the methods to the pages. This bean is used in later chapters to either set or retrieve values representing the state of the application.

Perform the following steps to create a Java class to define the attributes and hold the values representing the state of the application:

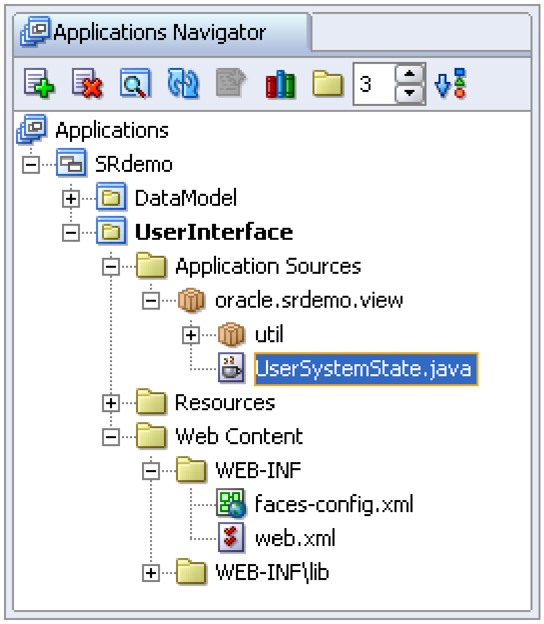
1. Select the **UserInterface** project. From the context menu, select **New**.
2. In the New Gallery select the **General** category (if it is not already selected), and choose **Java Class** in the Items pane. Click **OK**
3. Type **`UserSystemState`** as the class name and **`oracle.srdemo.view`** as the Package. Leave all other fields at their defaults.
4. The basic class is created and ready for coding. To save time, open the **`UserSystemState.txt`** file (found in the `<tutorial_setup>\files` directory or wherever you unzipped the setup file) and copy its code. Then paste it into the newly created class. Save your work.

The methods in this class are used to determine the specific information about the user who is navigating through the application. As we discussed earlier, the information includes the user ID, user status, current service request, and more. You need to expose the methods in the class for them to be easily available to your application. You do that by creating a managed bean, as described in the next steps.

5. If the `faces-config.xml` file is not already open, select **UserInterface > Web Content > WEB-INF** in the Applications Navigator and then open the file.
6. Click the **Overview** tab to access the Managed Beans page.
7. With the **Managed Beans** category selected, click **New**.
8. In the Create Managed Bean pane, set the property values to those in the following table:

Field	Value
Name	`userState`
Class	`oracle.srdemo.view.UserSystemState` You may type the value or select it by clicking the Browse button and expanding the Hierarchy tab.
Scope	`session`
Generate Class If It Does Not Exist	Clear the check box.

9. The following screenshot depicts using the Hierarchy tab when defining the managed bean class.



10. Click **OK** to continue.

You're now ready to set and retrieve all the values you need to determine the state of the application.

Creating a Standard Look and Feel

When developing Web applications, you can provide a consistent user experience by maintaining a common look and feel (same use of color, same screen design and layout) and—more importantly—by creating similar interaction behaviors. You can create a simple template that can be used as the basis for each of the pages in the application. This can increase your productivity as a developer because you do not need to "reinvent the wheel" for each page you build.

Perform the following steps to create a template that serves as the basis for developing all the pages in the SRDemo application:

1. In the Navigator, select the **UserInterface** project, and then choose **New** from the context menu.
2. In the New Gallery, expand the **Web Tier** node in the Categories pane (if it is not already expanded) and choose **JSF**.
3. In the Items pane, choose **JSF JSP** and then click **OK**. The Create JSF JSP Wizard launches.
4. Complete the first three steps of the wizard using the following values:

Wizard Step 1: JSP File

Field	Value
File Name	`SRDemoTemplate.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the

	<jdev_install>\`SRDemo\UserInterface\public_html\`**`Template`** folder.
Type	`JSP Document`
Mobile	Clear the check box.

5. Click **Next**.

Wizard Step 2: Component Binding	
Field	**Value**
Do Not Automatically Expose UI Components in a Managed Bean	Ensure that this option is selected. (Because this is just the outline for the pages you are going to develop later, you choose not to create a managed bean for the UI components at this point. Later, when you create the detailed individual pages, you create managed beans for the pages' UI components.)

6. Click **Next**.

Wizard Step 3: Tag Libraries	
Field	**Value**
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

7. Click **Finish**

   The empty page appears in the Visual Editor.

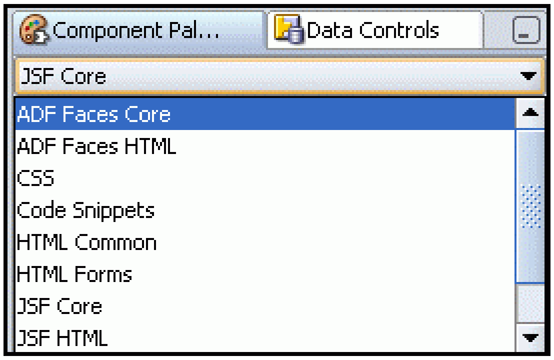
Adding Components to the Page

The next steps show you how to add components to the template page. The Component Palette is used for creating components in the different visual editors.

The Component Palette comprises a number of palette pages. Each palette page contains a logical grouping of components. For example, when you created the JSF navigation diagram, the palette page contained JSF Page, JSF Navigation Case, Note, and Attachment. Those are the items that are appropriate for a JSF navigation diagram.

In some cases there are multiple sets of components available. When that is the case, you select the group of components that you want by clicking the list of palette pages on the component palette.

The following screenshot shows some of the palette pages you use to create your JSF pages:



If the Component Palette is not visible, select **View > Component Palette**.

1. Select the **ADF Faces Core** page in the Component Palette, and then locate the **PanelPage** component in the list.
2. Drag the **PanelPage** to the template in the Visual Editor.
3. In the Property Inspector, type **`Change me`** in the Title property. This title will be modified appropriately for the pages you subsequently develop from the template page.
4. Add branding to the page as follows: Locate a directory called **`images`** in the `<ADFTutorialSetup>\files` directory. Using Windows Explorer (or the equivalent in your operating system), copy it to `<jdev_install>\jdev\mywork\SRDemo\UserInterface\public_html`.
5. In the Application Navigator, select the **Web Content** folder and choose **Refresh** from the View menu.
6. Locate the **`images`** node, expand it if it is not already expanded, and drag **`SRBranding.gif`** to the "branding" facet at the upper left of the template page. In the pop-up window, choose **`GraphicImage`** as the type of component to create.

In the next steps, you add a `LoadBundle` tag, which is used to help in the translation process when internationalizing applications. The `loadBundle` tag identifies the resource bundle that is used in the jspx pages of the application (that is, the `UIResources.properties` file you copied in the previous section).

7. Select the **JSF Core** Component Palette page. Drag the **`LoadBundle`** component to the Structure window, and drop it above `<>afh:head`.
8. In the Insert LoadBundle pop-up window, set the Basename to **`UIResources`** (or use **[...]**) and click the Properties File radio button, to browse for the properties file. Remember that the file is in `<jdev_install>\jdev\mywork\SRDemo\UserInterface\`. Set the Var property to **`res`** (`res` is the page-scoped alias for the resource bundle, which can then be used in Expression Language throughout the pages). Click **OK.**
9. Add copyright information as follows: In the Visual Editor (or Structure window), select the **appCopyright** facet. A facet is a slot in the panelPage into which you can place a UI component to provide a particular visual effect on the page. Right-click the facet and choose **Insert inside AppCopyright**, and then choose **OutputText** from the context menu.

10. With af:outputText still selected in the Structure window, choose **Properties** from the context menu. Click **Bind** to the right of the Value field.
11. In the “Bind to Data” dialog box, expand the **JSP Objects** node and then the **res** node in the Variables tree.
12. Scroll down to locate **srdemo.copyright**. Select it and click **>** to shuttle it into the Expression pane. Click **OK**, and then click **OK** again. (Alternatively, you can type **`#{res['srdemo.copyright']}`** directly into the Value property in the Property Inspector.)

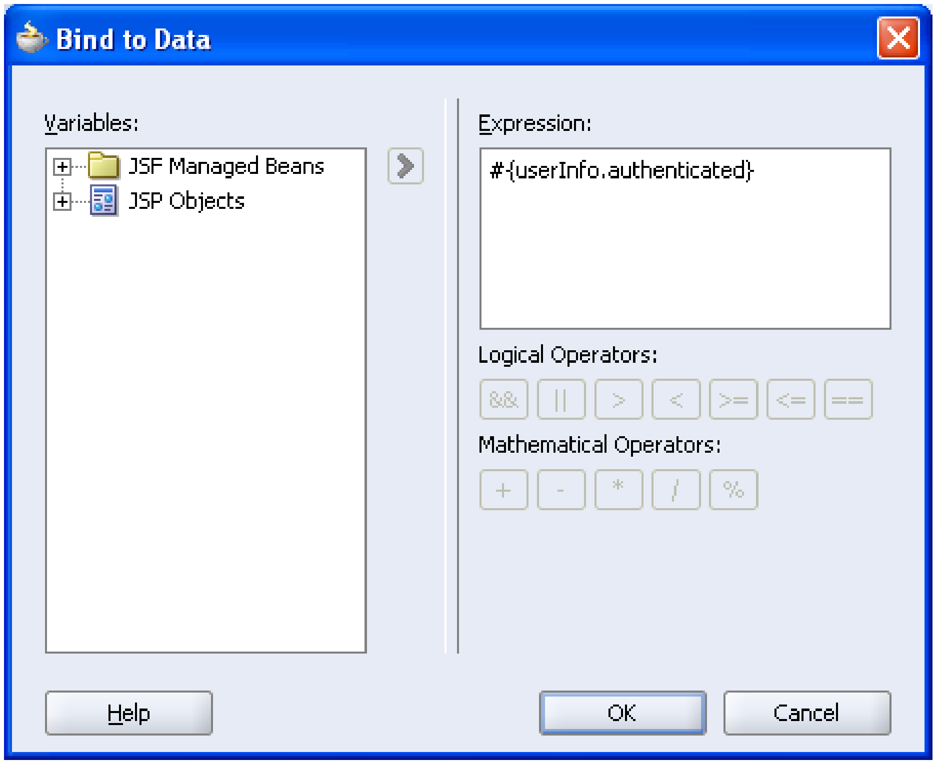
    The screenshot in the “Providing for Translation of the User Interface” section shows this as one of the name=value pairs in the `UIResources` file.
13. Change the Escape property to **false** so that the `©` markup in the resource string is printed as a correct copyright symbol.
14. Add an About link to provide information about the application as follows: Select the **appAbout** facet in the Visual Editor. From the context menu, choose **Insert inside appAbout** and then choose **CommandLink**.
15. Select the commandLink in the Structure window, and choose **Properties** from the context menu.
16. In the CommandLink Properties dialog box, click **Bind** to the right of the Text property.
17. In the “Bind to Data” dialog expand **JSP Objects** and then **res** in the Variables tree.
18. Scroll to locate **srdemo.about** and click **>** to shuttle it into the Expression pane. Click **OK**, and then click **OK** again**.** (Alternatively you can type **`#{res['srdemo.about']}`** in the Text property in the Property Inspector.)
19. Set the Immediate property to **true**.

    You use the Immediate property to shortcut the JSF lifecycle when you don't need the data in the screen to be applied, as with a Help link (or here with the About button).
20. Add a label to display information about the currently logged-in user, as follows: Select the **InfoUser** facet in the Structure window (expand the **PanelPage facets** node to locate it), and from the context menu, choose **Insert inside infoUser** and then choose **JSF HTML**. From the Insert JSF HTML Item dialog box, choose **Output Format**. Then click **OK**.

    Output Format is a standard JSF HTML tag and is used to display a localized message
21. In the Structure window, right-click **h:outputFormat**, and then choose **Properties** from the context menu. In the Output Format Properties dialog box, click the **Value Bind** button.
22. In the Bind to Data dialog box, expand the **JSP Objects** node and then the **res** node in the Variables tree.
23. Scroll through the list to locate **srdemo.connectedUser**. Select it and click **>** to shuttle it into the Expression pane. Click **OK.**

24. While still in the Output Format Properties window, click the **Rendered Bind** button, and in the Expression field, type **`#{userInfo.authenticated}`**. Click **OK**, and then click **OK** again.

    The userInfo managed bean provides access to security information obtained from the container security. Here you are checking that the current session is actually authenticated. You have not yet created the userInfo managed bean; you do so in a later chapter.

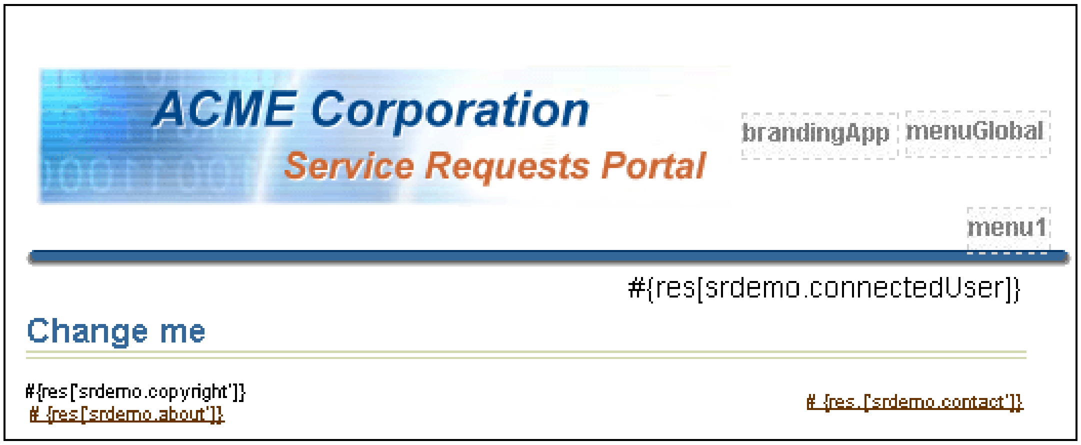


25. Define a parameter to pass in the name of the logged-in user, as follows: Right-click **`h:outputFormat`** in the Structure window, and choose **Insert inside h:outputFormat→Param** from the context menu. In the Insert Param dialog box, type **`#{userInfo.userName}`** in the Value field. Click **OK.**

26. Add contact information as follows: Locate the **appPrivacy** facet in the Structure window. Then right-click and choose **Insert inside appPrivacy→CommandLink** from the context menu.

27. In the Property Inspector, type **`#{res['srdemo.contact']}`** in the Text property and **`dialog:globalContact`** in the Action property. Note that this format (`dialog:`) is a feature of ADF Faces and is not provided by the JSF specification.

28. Save `SRDemoTemplate`. Your template page should look like the following screenshot:



Summary

In this chapter, you incorporated some standards into the SRDemo application. To accomplish this, you performed the following key tasks:

- Created files to hold commonly used utilities
- Created a resources file to facilitate translation of the application's UI
- Created a template page to provide a standard look and feel for application pages

5

Developing a Simple Display Page

This chapter describes how to create the SRList page, a simple display page at the center of the SRDemo application that enables users to view information about service requests.

The chapter contains the following sections:

- Introduction
- Creating the Page Outline
- Adding User Interface Components to the Page
- Wiring Up the Edit Button
- Wiring Up the View Button
- Defining Refresh Behavior
- Adding a Menu Bar to the Page
- Adding a Drilldown Link
- Running the Page
- Summary

Introduction

The SRList page is at the center of the SRDemo application. It is the first page that users see after logging in, and they can navigate from this page to all the other pages in the application.

Note: If you did not successfully complete Chapter 4, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter5` to hold the starter application. If you used the default settings, it should be in `<jdev-install>\jdev\mywork\Chapter5`.
2. Unzip `<tutorial-setup>\starterApplications\SRDemo-EndOfChapter4.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select `<jdev-install>\jdev\mywork\Chapter5\SRDemo\SRDemo.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 4.

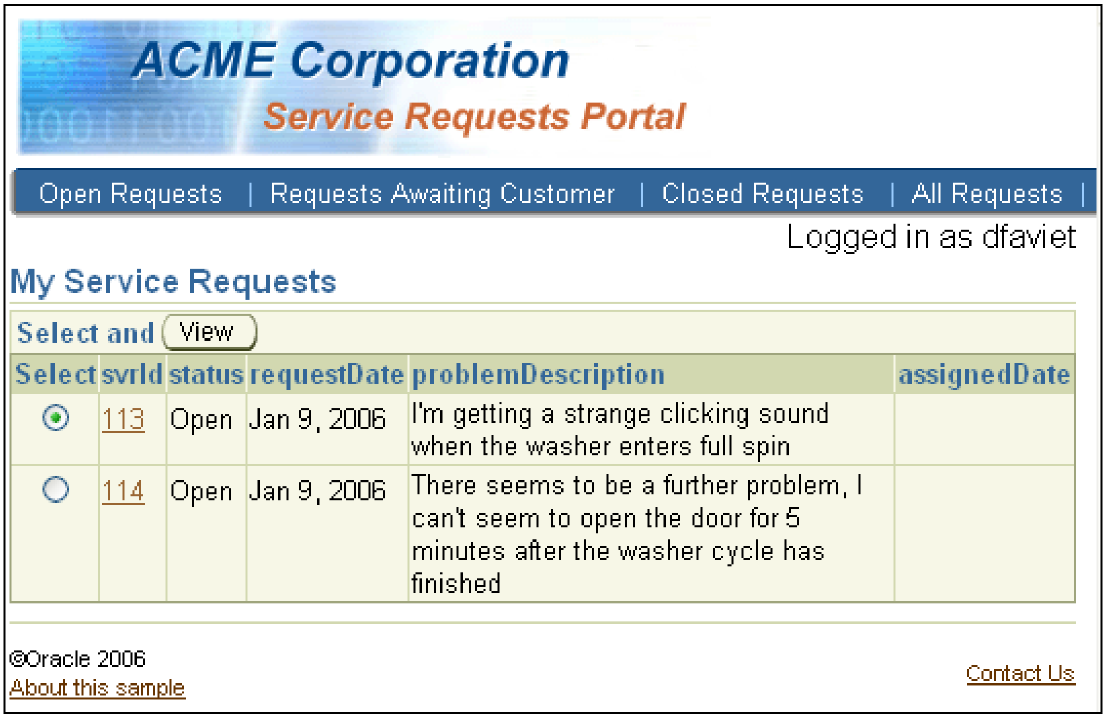
Double-click the `faces-config.xml` file in the Navigator to revisit the page-flow diagram that you created and see how the SRList page relates to the other pages.

Here are some points to note about the SRList page:

- The page displays each existing service request and its status.
- Any user (customer, technician, or manager) can access the page.
- When a customer logs in, all service requests pertaining to that customer are displayed. A View button is available.
- When a technician logs in, all service requests assigned to that technician are displayed. View and Edit buttons are available.
- The list of displayed requests can be filtered by clicking the second-level menu to show one of the following: all service requests currently open, only requests with a status of closed, or all requests irrespective of status. The default is requests that are open.
- Any user can select a service request and, by clicking the View button, go to the SRMain page to update the history of that request. In addition, technicians can click the Edit button to visit the SREdit page, where they can edit the request.
- To add a new request, any user can navigate to the SRCreate page by choosing the Create New Service Request menu option.
- Technicians can navigate to the Search page (SRSearch) by using the Advanced Search menu tab.

- Managers can see all service requests. All menu options are available to managers.

The following screenshot shows you how the finished SRList page should look:



To create the SRList page with the functionality and look-and-feel described in the preceding list and screenshot, you now perform the following key tasks:

- Creating the page outline, based on the template page you created in Chapter 4
- Adding user interface components to the page
- Wiring up the View and Edit buttons
- Defining Refresh behavior
- Creating a menu bar to enable users to view service requests with different statuses or to create a new service request
- Creating drilldown functionality to enable users to select a row in the table and navigate to the SRMain page to add information for the selected service request

Creating the Page Outline

In this section, you create the SRList page and add the template to apply the appropriate look and feel.

1. If it is not already open, double-click the `faces-config.xml` file to view the Page Flow Diagram.
2. Double-click the **SRList** page to invoke the JSF Page Wizard.
3. Complete the first three steps of the wizard using the following values:

Wizard Step 1: JSP File	
Field	**Value**
File Name	`SRList.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRDemo\UserInterface\public_html\app` folder.
Type	`JSP Document`
Add Mobile Support	Clear the check box.

4. Click **Next**.

Wizard Step 2: Component Binding	
Field	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that the option is selected.
Name	`backing_app_SRList`
Class	`SRList`
Package	`oracle.srdemo.userinterface.backing.app`

5. Click **Finish** to create the page details. The new SRList page is displayed in the Visual Editor.
6. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, shrink the **afh:html** node and select it. From the context menu, choose **Copy**.
7. Click the tab to return to the SRList page, and in the Structure window expand the **`f:view`** node.
8. Delete the **`html`** node. Then right-click **`f:view`** and choose **Paste** from the context menu. The look and feel that you created earlier is now applied to the new page.

Adding User Interface Components to the Page

Perform the following steps to add some user-interface elements to the page:

1. Add a title to your page as follows: Click the page in the Visual Editor to select it. (Alternatively, you can select **`af:panelPage`** in the Structure window.) In the Structure window, choose **Properties** from the context menu.
2. In the PanelPage Properties dialog box, click **Bind** in the Title property.
3. In the Bind to Data dialog box, expand the **JSP Objects** node and then the **res** node in the Variables tree.

4. Scroll through to locate **srlist.pageTitle**. Select it and shuttle it into the Expression pane. Click **OK**, and then click **OK** again.

   This is the name of the page title resource as defined in the `UIResources.properties` file that you created in the preceding chapter. The value displayed at run time is the actual title of the page.

5. Add a header to display in the browser title when you run the page, as follows:
   In the Structure window, select **`afh:head`**. In the Property Inspector, set the Title property to **`#{res['srdemo.browserTitle']}`**.

   Note that you can set these values by invoking the Properties page and then the "Bind to Data" dialog box, and then picking from the Variables tree as you did in the preceding step. (Alternatively, you can type the value in the Property Inspector.)

6. You now add data to the page. The data is in the form of a read-only table based on the findServiceRequests data collection. First, ensure that the Data Control Palette is visible to the right of the Visual Editor. If it is not, choose **Data Control Palette** from the View menu.

7. Expand the **SRPublicFacadeLocal** node, and then choose **findServiceRequests(Integer, String)→ServiceRequests** from the list of data collections.

8. Drag the collection to the structure window and drop it on the af:panelPage. From the Create pop-up menu, choose **Tables→ADF Read-only Table**.

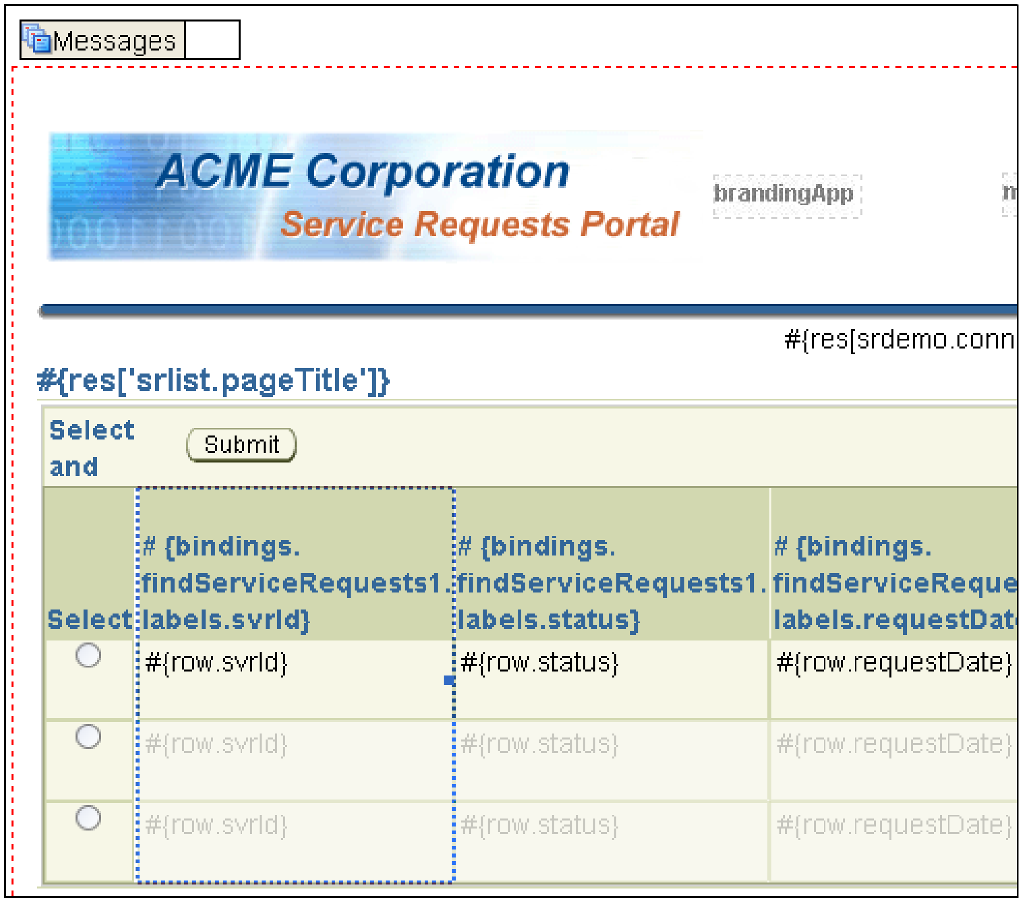
9. In the Action Binding Editor, set the value of the `filedBy` parameter to **`${userInfo.userId}`** and the value of the `status` parameter to **`${userState.listMode}`**. Click **OK**.

10. JDeveloper issues an alert window that it has added Client Project Libraries.
    Click **OK** to close the alert.

    Note: You haven't yet created the `userInfo` class and managed bean; you do so in the next chapter.

11. In the Edit Table Columns dialog box, reorder the columns so that **svrId** is at the top of the list, followed by **status, requestDate, problemDescription**, and **assignedDate**. This determines the order of the columns on the page. Make sure that the "Component to Use" column has **ADF Output Text** set for every column. Select the **Enable Selection** check box to add an option selection column to the table, and then click **OK**.

12. With the table selected in the Visual Editor, change the Id property in the Property Inspector to **`srtable`**. Save the page. At this point, your page should look like the following screenshot:



Wiring Up the Edit Button

The SRList page provides a starting point for customers and technicians. From this page, users can navigate either to an edit page or to a view page. The Edit button is available only when a technician or manager logs in. Clicking this button takes users to the SREdit page, where details of the currently selected service request can be modified.

Perform the following steps to change the default submit button to the Edit button.

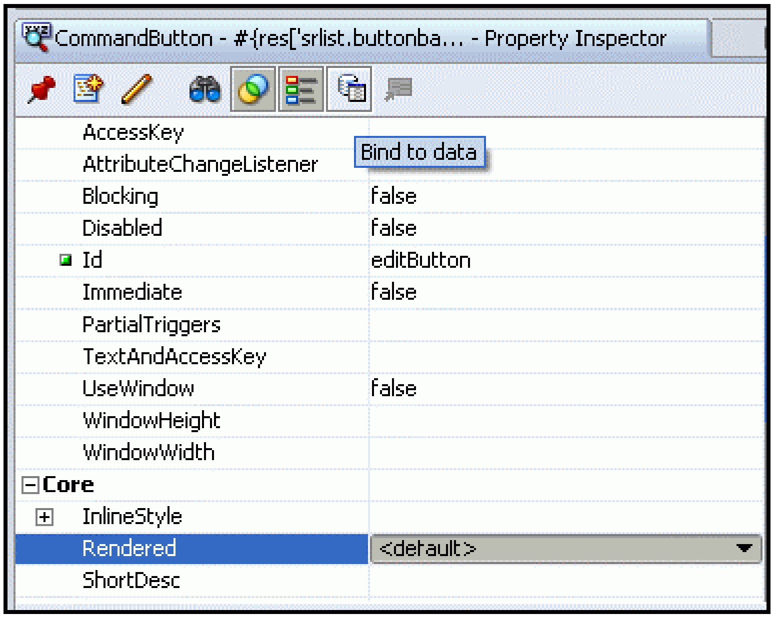
1. Click the **Submit** button. In the Property Inspector, change the Text property to **`#{res['srlist.buttonbar.edit']}`**.

   This is the name of the button resource as defined in the `UIResources.properties` file. The value displayed at run time is "Edit."

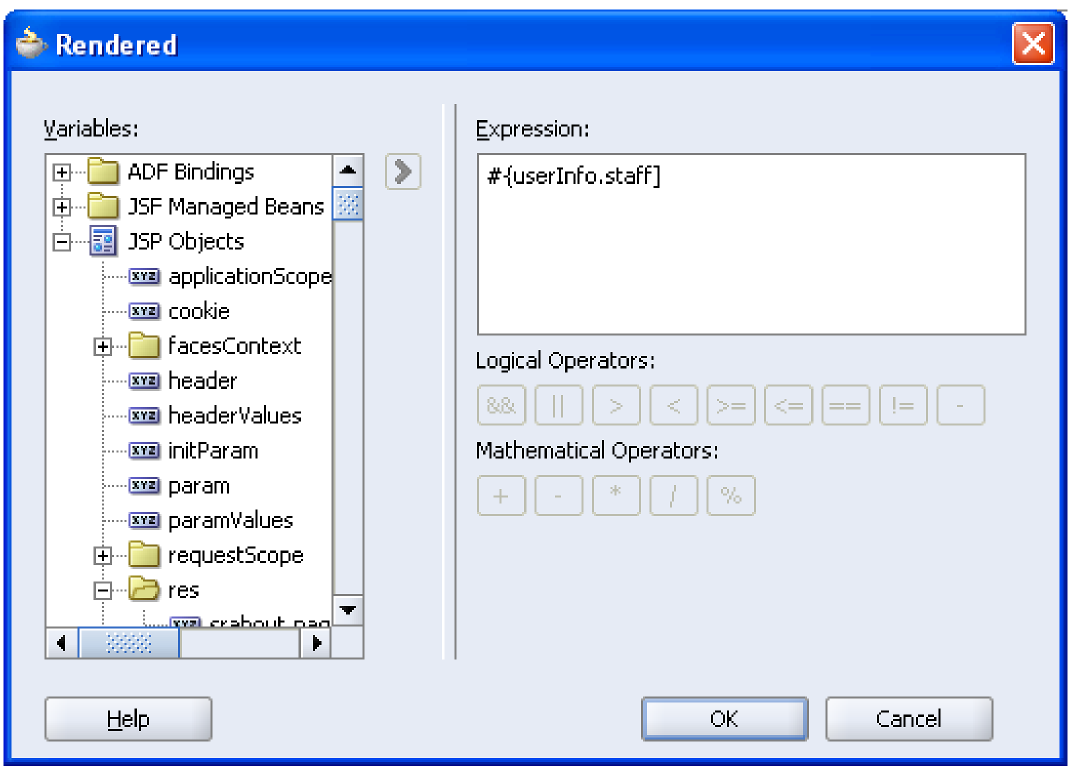
2. Also in the Property Inspector, set the Id property to **`editButton`**.

   Notice that JDeveloper searches all of the source code for uses of the original ID of this button. It automatically changes all of those references to the new name.

3. You now specify that you want to display the Edit button only when the user who is currently logged in is a member of staff, as follows: Select the **Rendered** property, and then click the **Bind to Data** button in the toolbar of the Property Inspector (It is the second button from the right; use the tooltips labels to check which button you need.)



4. In the Rendered dialog box, type **`#{userInfo.staff}`** in the Expression field. Click **OK**. (You have not yet created this class and managed bean; you do so in the next chapter.) This is to verify that the logged-in user is indeed a member of staff.



5. Create a method in the backing bean that passes the ID of the currently selected service request through to the Edit page, so that the appropriate record can be retrieved and displayed. In the Visual Editor, double-click the **Edit** button to invoke the backing bean.

6. In the Bind Action Property dialog box, click **OK** to add the `editButton_action` method to the backing bean.

7. In `editButton_action`, you need to add code to set the service request ID to the current row and return the name of the JSF navigation case that you want to use. When you set the return to a navigation case name, JSF takes that value and forwards the user to the end page of that navigation case. Add the following code to the `editButton_action` method:

```
setCurrentSvrIdFromRow();
return "edit";
```

8. Because the `setCurrentSvrIdFromRow` method does not yet exist, JDeveloper will flag it as a code error. Click the **CodeAssist** icon (the light bulb in the left margin) to create the method.

9. Implement the `setCurrentSvrIdFromRow` method by adding the following code to the method you just created. (Press [Alt] +[Enter] to import the appropriate package when prompted by Code Assist.)

   This code does two things:

   - It retrieves the current row service request ID and stores it in the UserState managed bean.
   - It sets the navigation path to return to the SRList page on completion of the edit.

```
FacesContext ctx = FacesContext.getCurrentInstance();

JUCtrlValueBindingRef tableRowRef =
    (JUCtrlValueBindingRef) this.getSrtable().getRowData();

Integer svrId =
    (Integer)tableRowRef.getRow().getAttribute("svrId");


UserSystemState.storeCurrentSvrID(svrId);


//Store away where we want to come back to

UserSystemState.storeReturnNavigationRule
  ("globalHome");
```

10. Still in the backing bean, add a new class variable as follows:

```
private BindingContainer bindings;
```

   (Press [Alt] +[Enter] to import the `oracle.binding` package when prompted by Code Assist.)

11. Because you are creating the page with auto-binding off, you need to add the bindings manually in this step. Right-click **`bindings`**, and then select **Generate Accessors** from the context menu. In the Generate Accessors menu, click **OK** to generate both **`setBindings`** and **`getBindings`** methods.

12. Save the **`SRList.java`** file.

13. You click the page and select **Run** from the context menu to run the page. However, you will see ”no rows yet” displayed in the table. This is because the page is based on the

service requests for a specific logged-on user. You have not yet set the application to use any type of logon, so the page does not return any rows.

Wiring Up the View Button

The View button is available to all users of the application, enabling them to navigate to the SRMain page to update the history of a selected request.

Perform the following steps to specify the View button functionality:

1. Return to `SRList.jspx` in the Visual Editor.
2. Select the **Edit** button you just created.
3. Right-click and select **Insert After Command Button → Command Button** from the context menu.
4. In the Property Inspector, set the Text property to `#{res['srlist.buttonbar.view']}` and the Id property to `viewButton`.
5. As with the Edit button, you need to create a method in the backing bean that specifies exactly what must happen when the View button is clicked. Double-click the **View** button to invoke the backing bean for the page. In the Bind Action Property dialog box, click **OK** to add the `viewButton_action` method to the backing bean.
6. In the `SRList.java` file, add the following code to the `viewButton_action()` method:

   ```
   return drillDown_action();
   ```

7. The `drillDown_action` method does not yet exist, so click the **Code Assist** icon to create it.

   This method again uses the `setCurrentSvrIdFromRow` method that you created in the previous section. It passes the ID of the currently selected service request through to the SRMain page, so that the appropriate record can be retrieved and displayed. You use this method again at a later point in the page where you drill down on a service request.

8. Add the following code to the method to specify this behavior:

   ```
   setCurrentSvrIdFromRow();
   return "view";
   ```

9. Save the file.

Defining Refresh Behavior

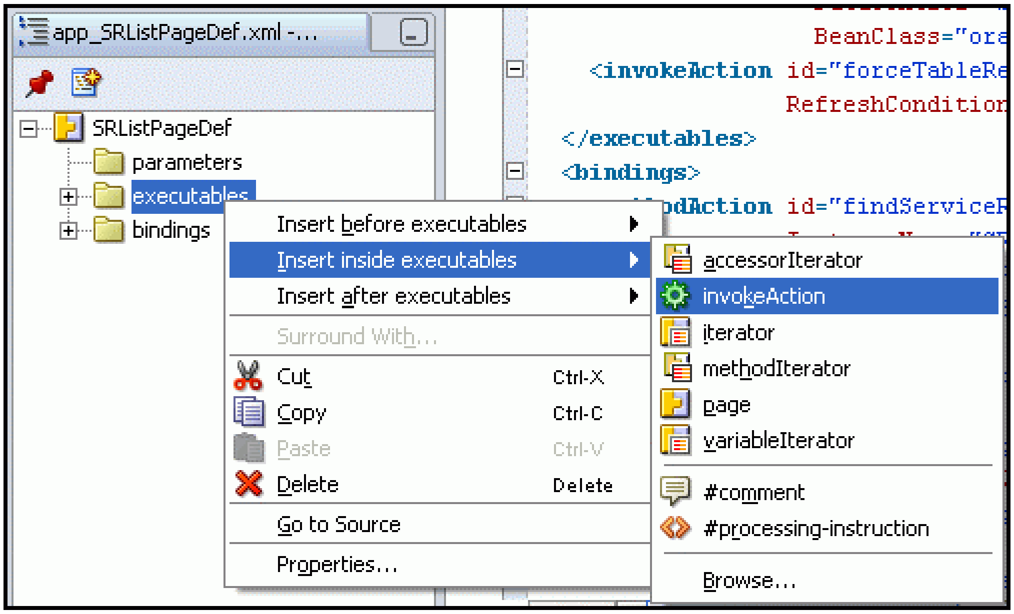
Users can return to the SRList page from anywhere in the application. In the steps below, you create the `userState.refresh` parameter to determine whether the List page needs to be refreshed on returning from another page.

If you want to force the page to refresh (for example, when data has been updated), you set this parameter in the UserState bean to True, and Expression Language then picks this up and executes the queries on the List page. The exception to the refresh is if the page is invoked from a JSF postback. A postback occurs when the page is being refreshed because of a user action on the page. If this occurs, you do not need to force a refresh of the page.

Perform the following steps to specify refresh behavior for the page:

1. Return to the SRList page. In the Structure window, right-click `af:panelPage` and then choose **Go to Page Definition** from the context menu.

2. In the Structure window, expand the **SRListPageDef** node if it is not already expanded. Right-click **executables**, and then choose **Insert inside executables→invokeAction** from the context menu.



3. Set the ID to `forceTableRefresh`.

4. In the Binds field, click the arrow and choose **findServiceRequests**.

5. Click the **Advanced Properties** tab**,** and click **[...]** in the **RefreshCondition** property.

6. In the Advanced Editor, copy the following line of code into the expression:

   ```
   ${(userState.refresh) and (!adfFacesContext.postback)}
   ```

   This expression adds the proviso that if the page is called as part of a postback from this same page, then a refresh should not take place.

7. Click **OK**, and then click **OK** again.

8. Save the page.

Adding a Menu Bar to the Page

The SRList page has a set of menu options so that users can choose to view service requests with a status of open, closed, or pending, or they can view all requests regardless of status. There is also a link to create a new service request.

Perform the following steps to add these menu options to the page:

1. You first create a second-level menu bar to hold the menu options. In the Structure window, expand the **PanelPage facets** node, and scroll down to **menu2**. Right-click, and from the menu choose **Insert inside menu2→MenuBar**.

2. In the **Data Control Palette** expand the **SRPublicFacadeLocal** node and select **findServiceRequests(Integer, String)**. Drag it to the Structure window onto **af:menuBar**. In the Create pop-up menu, choose **Methods→ADF Command Link**.

3. In the Property Inspector, type `#{res['srlist.menubar.openLink']}` in the Text property.

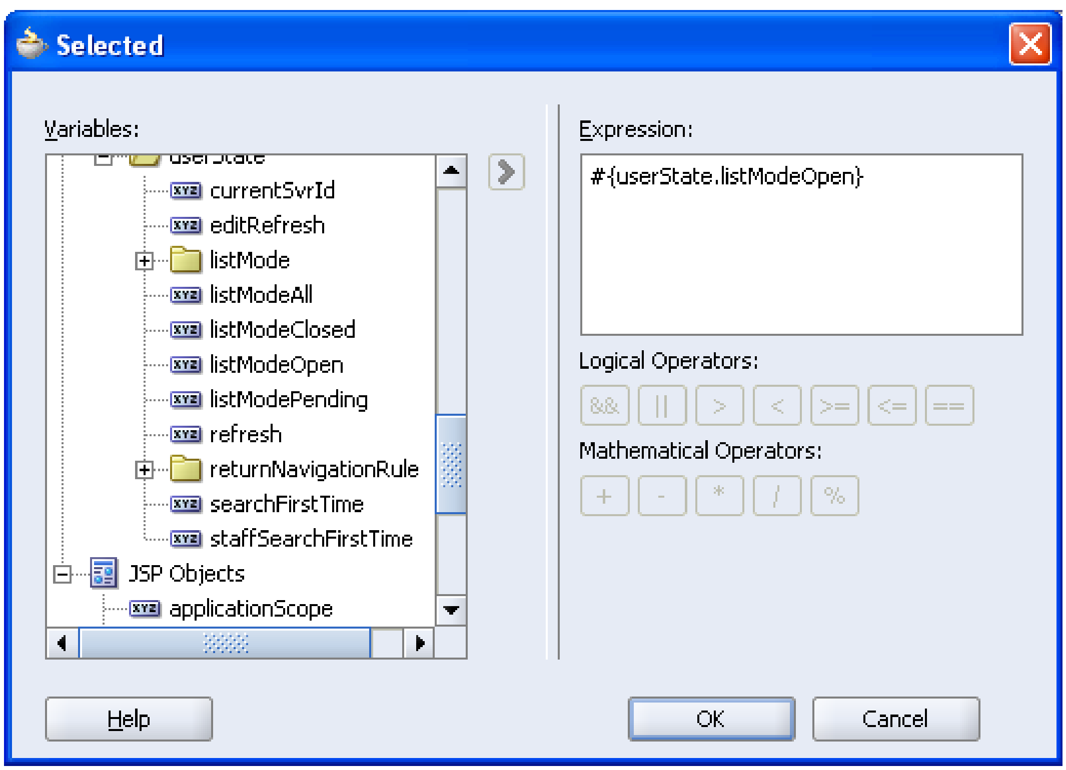
4. Repeat step 2 an additional three times, creating a total of four menu options. Set the Text property for each link as follows:

```
#{res['srlist.menubar.pendingLink']} #{res['srlist.menubar.closedLink']}
#{res['srlist.menubar.allRequests']}
```

5. Convert each of the command links to command menu items so that they appear as options in the menu bar, as follows: Right-click each of the command links, and then choose **Convert** from the menu. In the Convert CommandLink dialog box, select **CommandMenuItem** and click **OK** to change these elements to menu items.

   To make the selected tab appear highlighted, use Expression Language to check for the status parameter being used for the links. There is a convenience function (`isListModeOpen`) defined in the userState managed bean that you created in the previous lesson.

   To do this for requests with a status of open (the default), click in the **Selected** property for the openLink menu item, and then click the **Bind to Data** button in the Property Inspector toolbar (second button from the right). In the Selected dialog box, expand the **JSF Managed Beans** node and then the **userState** node. From the list, choose **listModeOpen**. Click **>** to shuttle it into the Expression pane. Then click OK.



6. Do the same for each of the other menu items, selecting values as defined in the following table:

Menu Item	Expression
Pending Requests	`#{userState.listModePending}`
Closed Requests	`#{userState.listModeClosed}`
All Requests	`#{userState.listModeAll}`

Set an actionListener on the Open command link. ActionListeners fire when an event occurs (for example, when the user clicks a link). When the actionListener fires with the `userState.listMode` parameter set to Open (as you defined in step 6), service requests with a status of Open are displayed.

7. In the Structure window, right-click **first command menu item** and, from the context menu, choose **Insert inside af:commandMenuItem - … → ADF Faces Core** In the Insert ADF Faces Core Item dialog box, choose **setActionListener**. Then click **OK**.
8. In the Insert SetActionListener dialog box, set **From*** to #**{'Open'}**, and **To*** to #**{userState.listMode}**.

   (You can type in the value, or you can click the **Bind** button and, in the Bind to Data dialog box, expand the **JSF Managed Beans** node and then the **userState** node. Select **listMode** from the list, and shuttle it into the Expression pane. Click **OK**.)
9. Click **OK**. Set actionListeners inside each of the other three command menu items in the same way, selecting values as defined in the following table:

Menu Item	From*	To*
Pending Requests	`#{'Pending'}`	`#{userState.listMode}`
Closed Requests	`#{'Closed'}`	`#{userState.listMode}`
All Requests	`#{'%'}`	`#{userState.listMode}`

10. Add a commandMenuItem to the menu bar to enable users to create a new service request, as follows: Select the **ADF Faces Core** page in the Component Palette, and then drag a **CommandMenuItem** to the menu bar.
11. In the Property Inspector, set the Text property to **`#{res['srlist.menubar.newLink']}`** and set the Action property to **`create`**.
12. Save the page.

Adding a Drilldown Link

Now we add a link to the first column, enabling users to drill down on a service request and navigate to the SRMain page, where they can add new information to it.

Perform the following steps to add a drilldown link to your page:

1. In the Data Control Palette, expand the **findServiceRequests(Integer, String)** node and then the **ServiceRequests** node. Scroll down to find **Operations** and expand that node as well.

2. In the list, select **setCurrentRowWithKey** and drag it to the **svrId** column of the table, next to the existing text. From the pop-up menu, choose **Operations→ADF Command Link**.

3. In the Action Binding Editor, change the value of the parameter to the link from `${bindings.setCurrentRowWithKey}` to **`#{row.rowKeyStr}`**. Then click **OK**.

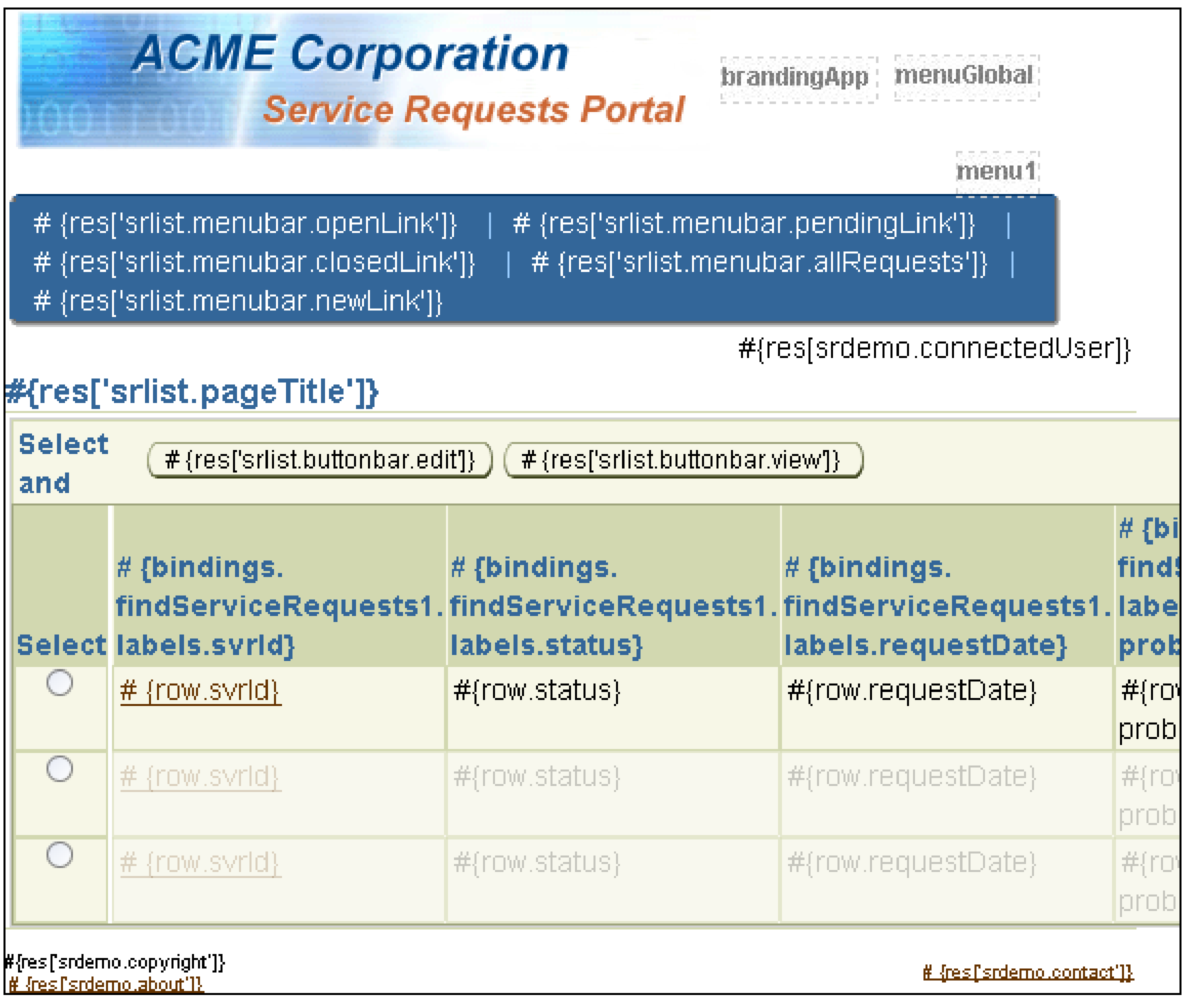
   Here, `row` is an index for the currently selected row, thus enabling you to get the key for that row. This key value is then passed to the `setCurrentRowWithKey` method. Therefore, the key value that the user selects by clicking the link is used to set the current row, which is then the row that is displayed when the user is forwarded by the link to the SRMain page.

4. Set the Text property of the command link to **`#{row.svrId}`** and set the Action property to use the **`drillDown_action()`** method you created earlier in this chapter.

5. In the Structure window, delete the original **`af:outputText`** with label #{row.svrId}.

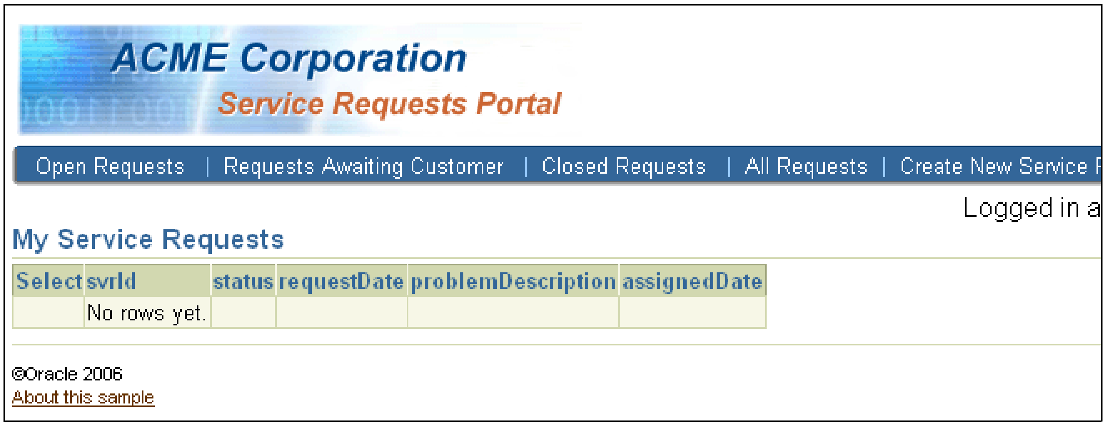
6. Save your page.

With the menu bar and drilldown link in place, the page should now look like the following screenshot:



Running the Page

Now that the page is finished, you can run it to see how it looks. With the SRList page open in the Visual Editor, right-click and select **Run**. When the page is displayed, it should look like the following screenshot:



Notice first that there is no data displayed. This is because you are running the SRList page directly. When the page is run as part of the application as a whole, a user ID is passed into the page from the logon so that service requests appropriate to the logged-on user can be displayed. You create logon functionality in a later chapter.

Notice the various UI components that you added to the page; note the values picked up from the `UIResources` file.

The menu tabs that you defined are displayed in the menu bar along the top of the page. You added highlighted text to the selected request tab. Because Open is the default request status, the Open Requests tab is bold.

The View and Edit buttons are not displayed. They appear when the table displays some data.

Summary

In this chapter, you created a display page to enable users of the SRDemo application to view information about service requests. To accomplish this, you performed the following key tasks:

- Created an outline page based on the template page you defined in Chapter 4
- Added user interface components to the page to display service request information
- Added View and Edit buttons for navigating to other pages in the application
- Specified refresh behavior to enable users to see service requests when returning from another page
- Added menuing to enable users to select a service request by status
- Created drilldown functionality to enable users to select a row in the table and navigate to the SRMain page

6

Implementing Login Security

This chapter describes how to build security for the SRDemo application. You first add authentication, enabling you to log in as a default user and see data on your pages. You then enable other users to access the application through their own user IDs and passwords.

The chapter contains the following sections:

Introduction

You should implement both authentication and authorization in the environment where the SRDemo application will run. Authentication determines which users get access to the application, and it is controlled by the roles assigned to each user. Authorization controls what type of behavior users are allowed to perform after they enter the application.

Security in SRDemo is based on J2EE container security. The available roles for the application are (all lowercase): user, technician, and manager

In the security container, the `remoteUser` value (container `userid`) matches the e-mail password in the application users table. The key security artifact is the `UserInfo` bean. This class reads the container security attributes when it is first referenced and then makes the key information available in a form that can be accessed via expression language.

For development purposes, test values for username and role can be injected into the `userInfo` object through managed properties in the `faces-config.xml` file. These settings are ignored if the application is deployed to a container that has security enabled, in which case the container security information is returned.

In this chapter, you build the authentication and authorization for the application, including:

- A class to manage the login and the role of a user
- A class to provide authentication for the user
- A named query to find users by their e-mail IDs
- Registering users who are allowed to access the application

---

Note: If you did not successfully complete Chapter 5, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named **`Chapter6`** to hold the starter application. If you used the default settings, it should be in **`<jdev-install>\jdev\mywork\Chapter6`**.
2. Unzip **`<tutorial-setup>\starterApplications\SRDemo-EndOfChapter5.zip`** into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select **`<jdev-install>\jdev\mywork\Chapter6\SRDemo\SRDemo.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 5.

Creating a Class to Control Security

The SRDemo application uses three core roles in determining who has access to perform what type of function. Each user must be classified with one of the three roles: user, technician, or manager. The default password for all users is `welcome`. All of these criteria are implemented using container-managed BASIC authentication provided by Oracle Application Server 10*g* or any other application server

The `UserInfo` bean is registered as a managed bean called `userInfo` that supports expressions such as #{userInfo.userName}. It returns either the login ID or the string `"Not Authenticated`."
The following table shows the expressions used in the `UserInfo` class and their values:

Expression	Value
`#{userInfo.userRole}`	Returns the current user's role in its string value (for example, manager)
`#{userInfo.staff}`	Returns true if the user is a technician or manager
`#{userInfo.customer}`	Returns true if the user is assigned a user role
`#{userInfo.technician}`	Returns true if the user is assigned a technician role
`#{userInfo.manager}`	Returns true if the user is assigned a manager role

The first set of tasks creates the containers you need and populates them with a test user and role. In this way, you can test the application. In the second set of tasks, you define users and roles in the Application Server, enabling users to log in with their own user IDs.

Creating a Class to Manage Roles

The first task is to create a class that contains the code needed to validate users, and to determine the available roles.

1. In the Applications Navigator, expand the **UserInterface** project. Right-click the **Application Sources** node and select **New**.
2. From the New Gallery, select the **Java Class** item from the General category.
3. Use the following table to populate the values in the Create Java Class pane:

Field	Value
Name	`UserInfo`
Package	`oracle.srdemo.view`
Extends	`Java.lang.Object`
Public (Attribute)	Select the check box.
Generate Default Constructor	Select the check box.
Generate Main Method	Clear the check box.

4. The basic class is created and ready for coding. To save time, open the **`UserInfo.java`** file (found in the `<tutorial-setup>\files\` directory or wherever you unzipped the ADFTutorialSetup file) and copy its code. Then paste it into the newly created class, replacing the generated code.

5. Click **Save** to save the file.

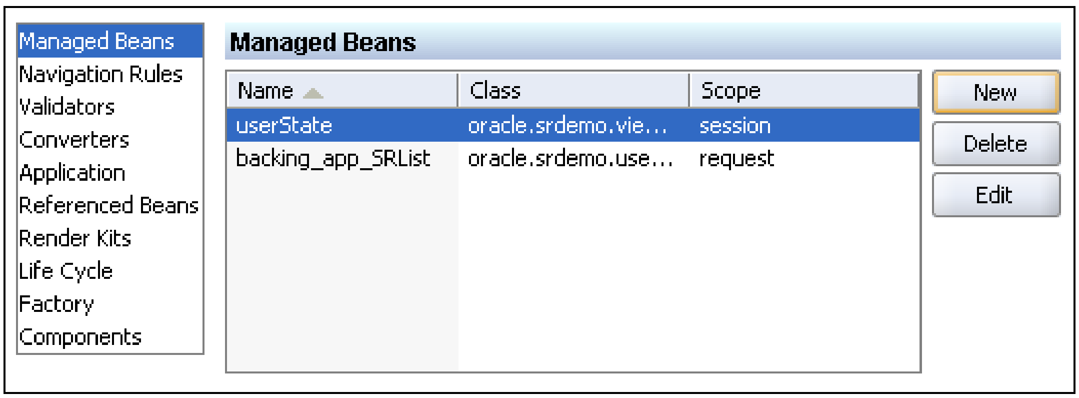
Creating a Class to Provide Authentication

The next task is to create and populate the managed bean. In this section, you create the `userInfo` managed bean, which enables the JSF pages to access the `UserInfo.java` class.

1. If the `faces-config.xml` file is not already open, right-click **UserInterface** and select **Open JSF Navigation**.

2. Click the **Overview** tab to expose a list of all the managed beans. You'll see a managed bean for each of the pages you created.

3. With the **Managed Beans** category selected, click **New**.



4. In the Create Managed Bean pane, set the property values to those in the following table:

Field	**Value**
Name	`userInfo`
Class	`oracle.srdemo.view.UserInfo` Type the value or select it by clicking the ellipsis (...) and expanding the Hierarchy tab.
Scope	`session`
Generate Class If It Does Not Exist	Select the check box.

5. Click **OK** to continue.

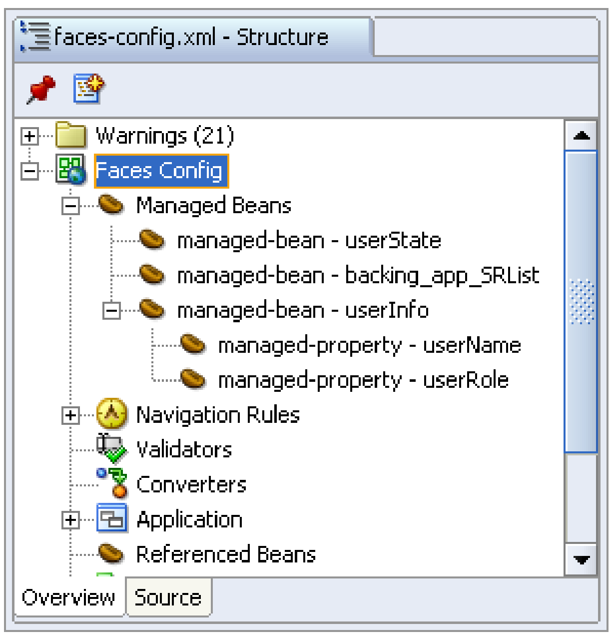
6. In the Structure window, right-click the new **userInfo** managed bean. Select **Insert inside managed bean | managed-property**. This step enables you to log on during testing with a default username. Use the following table to populate the required values:

Field	Value
Name	`userName`
Class	`java.lang.String` Type the value or select it by clicking the ellipsis (...) and expanding the Hierarchy tab.

7. Click **OK** to continue.
8. Create another managed-property inside the **userInfo** managed bean. Use the following table to populate the required values:

Field	Value
Name	`userRole`
Class	`java.lang.String` Type the value or select it by clicking the ellipsis (...) and expanding the Hierarchy tab.

9. Click **OK** to continue. Your Structure window should look like the following screenshot:



10. For testing purposes, double-click the **userName** managed-property and set the Value property to **`sking`**. Then click **OK**.
11. For testing purposes, double-click the **userRole** managed-property and set the Value property to **`manager`**. Then click **OK**.
12. Click the **Source** tab and examine the new code. Notice that the managed properties have been populated with your values.

```
<managed-bean>
  <managed-bean-name>userInfo</managed-bean-name>
  <managed-bean-class>oracle.srdemo.view.UserInfo</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>userName</property-name>
    <property-class>java.lang.String</property-class>
    <value>sking</value>
  </managed-property>
  <managed-property>
    <property-name>userRole</property-name>
    <property-class>java.lang.String</property-class>
    <value>manager</value>
  </managed-property>
</managed-bean>
```

You have now added a user so that the pages can run and be populated with data. This is sufficient for testing purposes, but eventually you will need to provide security for the application.

Integrating a User with the Application

When you deploy the application, you will want to enable users to provide their own information and have that information recorded. The application then needs to be aware of who is logged in so that it can decide what menus and functionality to provide. The user's e-mail ID is passed from the security container to the application to control the type of access available to the user (for example, access to the SRList and SRCreate pages).

Creating a Named Query to Manage Users

You can identify the user who logs in to the application through a named query. The query is read-only and takes a String parameter containing the e-mail ID. This query returns a user object for a particular e-mail ID received from the container security. You now create a named query in the Users descriptor of the `DataModel`→ `TopLink`→`SRMap` file.

1. Create a named query for the Users descriptor. The query is based on the e-mail ID and receives its value from the security container. Use the values in the following table. (For the detailed steps to create the query see chapter 2.)

Field	Value
Named Query Name	`findUserByEmail`
Type	`ReadObjectQuery`
Parameter Type	`java.lang.String`
Parameter Name	`emailParam`

2. Create an expression for the named query. Use the values in the following table to complete the definition:

Field	Value
First Argument Query Key	`email`
Operator	`EQUAL`
Second Argument - Parameter	`emailParam`

You have just declaratively created named query and now need to create the method in the session bean. You now include the method to retrieve the e-mail ID.

3. Convert the named query into a method to expose the value to the application. Expand the **DataModel→Application Sources→oracle.srdemo.datamodel** nodes.
4. Right-click the EJB Session bean and select **Edit Session Facade** to add the new query to the bean. Expand the **Users** node and select the check box in front of the **public Users findUserByEmail(String emailParam)**. Click **OK**.
5. Click **Save All** to save your work.
6. In the Applications Navigator, select the **`SRPublicFacadeBean.java`** node and, from the context menu, select **Create Data Controls**.

   This action pushes the current EJB forward to update the existing data control. The Data Control palette updates to include controls for any newly added methods.

Exposing the User E-mail ID to the Application

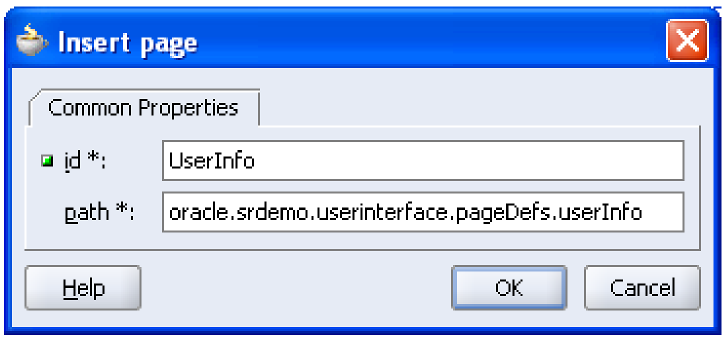
You now create a method to expose the TopLink Java object to the outside world. After the value is exposed, it can be used elsewhere in the application. You create an XML file to allow all the JSF pages to access the user IDs.

1. Create an XML file to contain the method. Expand the **UserInterface→Application Sources→oracle.srdemo.userInterface → pageDef** nodes, right-click, and select **New**.
2. Select **XML** from the General category and **XML Document** from the Items pane.
3. In the pop-up menu, name the file **`userInfo.xml`** and ensure that the directory includes your page definition path (…`SRDemo\UserInterface\src\oracle\srdemo\userInterface\pageDefs`).
4. Copy the code in the **`userInfo.xml`** file, which is found in your `<tutorial-setup>\files` directory. Replace the existing code with the code from the file. The code defining the `UserInfo.xml` file is stored with the other page definition packages and uses the `findUserByEmail` method in the `SRPublicFacade` session bean.
5. Close and reopen JDeveloper to get JDeveloper to recognize `userInfo.xml` as a page definition.

In addition to the XML definition, you need to be able to access it by name (`UserInfo`) from

DataBindings.cpx. The DataBindings.cpx file is created the first time you open a Web page from the Page Flow Diagram in the HTML Visual Editor. The .cpx file defines the Oracle ADF binding context for the entire application. The .cpx file provides the metadata from which the Oracle ADF binding objects are created at run time. The binding context provides access to the bindings across the entire application. You can edit this file in the Property Inspector to add or remove parameters and to alter the binding container settings. This can be done from the Structure window with the data bindings file:

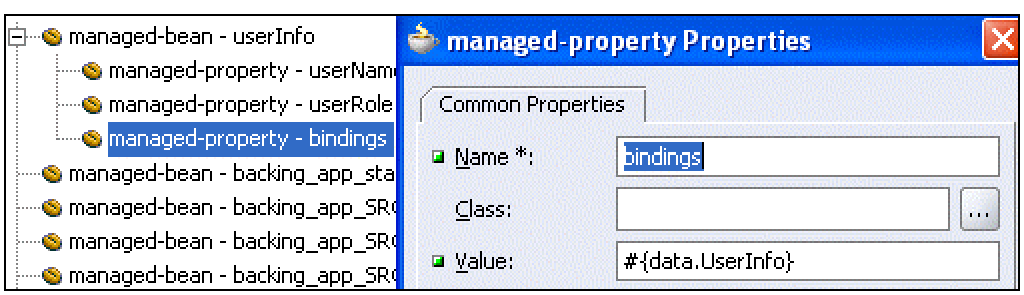
6. Expand the **UserInterface → Application Sources → oracle.srdemo.userInteface** nodes, and then select the **DataBinding.cpx** node.
7. In the Structure window, select the **pageDefintionUsages** node, and then choose **Insert inside pageDefinitionUsages | page**.
8. In the Insert page pop-up menu, set the ID to **`UserInfo`** and set the path to **`oracle.srdemo.userinterface.pageDefs.userInfo`**. Click **OK** to continue.



At run time, a reference to data.UserInfo now resolves to this binding definition, allowing the page to access the UserInfo class.

In the next steps, you assign a value to a managed property to point the userInfo.xml file you just created. This enables the managed bean to accept values from the return of the findUserByEmail named query.

9. In the Visual Editor, open the **`faces-config.xml`** file. Then click the **Overview** subtab and select the **userInfo** managed bean. Create a New Managed Property called **`bindings`**. The binding allows any reference to find the correct userInfo.
10. After it is created, double-click the **`bindings`** property and set the value property to **`#{data.UserInfo}`**. The following screenshot shows all the managed properties for the userInfo bean and the value for the bindings property:



Setting Up Container Security

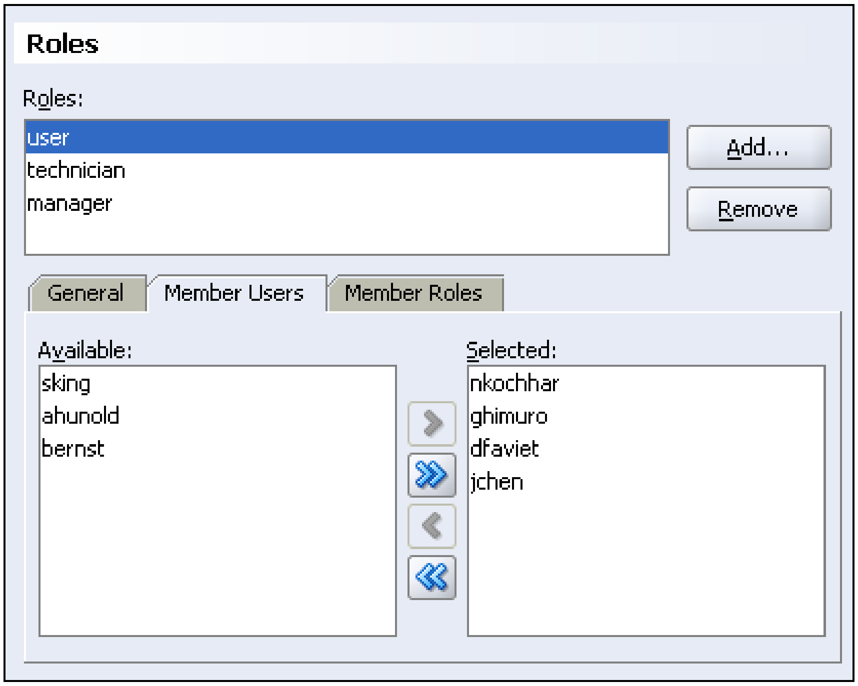
For users to be authenticated, they need to be registered in the JAZN-based security system of the J2EE container. Perform the following steps to register roles, usernames, and credentials in the `jazn-data.xml` file.

1. If its running, stop the OC4J Server by selecting **Run > Terminate >| OC4J**.
2. Navigate to **Tools > Embedded OC4J Server Preferences**.
3. Drill down through **Current Workspace(SRDemo) > Authentication(JAZN) > Realms**.
4. If the `jazn.com` realm does not exist, click the **New** button and name the new realm **`jazn.com`**.

   The realm is the general component of the security. The SRDemo component added later is a way to use the realm. In this way, you can use the same realm for many different applications.
5. Select the **Roles** node, and then click the **Add** button.
6. Create three roles: **`user`**, **`technician`**, and **`manager`**.
7. Create users with the values in the following table. Use the credential **`welcome`** throughout, and keep the usernames all lowercase. Then assign the usernames to their roles by selecting a role and clicking the **Member Users** tab. Assign each user to the corresponding role on the Selected side.

Username	Role
`sking`	`manager`
`nkochhar`	`user`
`ghimuro`	`user`
`ahunold`	`technician`
`bernst`	`technician`
`dfaviet`	`user`
`jchen`	`user`

The following screenshot shows the members assigned to the `user` role:



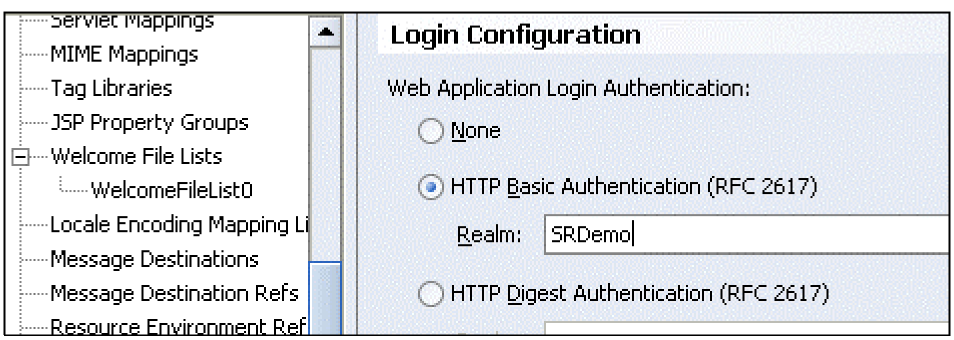
All of the data you entered is kept in a file at the root directory of the application. The convention used to name the file is `<applicationName>-jazn-data.xml`. In the tutorial, the file is named `SRDemo-jazn-data.xml`.

Setting Up Application Access

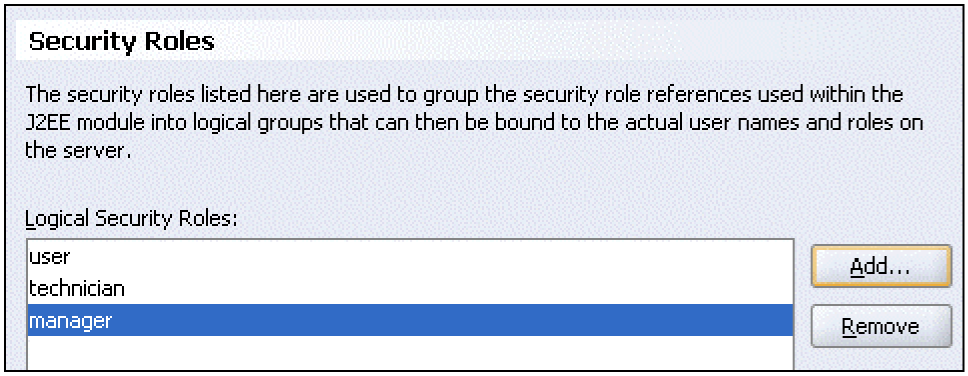
In this section, you define which roles that get access to which directory structures in the application.

1. In the Applications Navigator, expand the **UserInterface→Web Content→WEB-INF** nodes and select **web.xml.** Select **Properties** from the context menu.

2. Scroll down the left pane and select the **Login Configuration** node. Select the **HTTP Basic Authentication** option, and then set the Realm property to **`SRDemo`**.



3. Select the **Security Roles** node and add the same three roles as before: **`user`**, **`technician`**, and **`manager`**. (Click Add for each role you need to add).



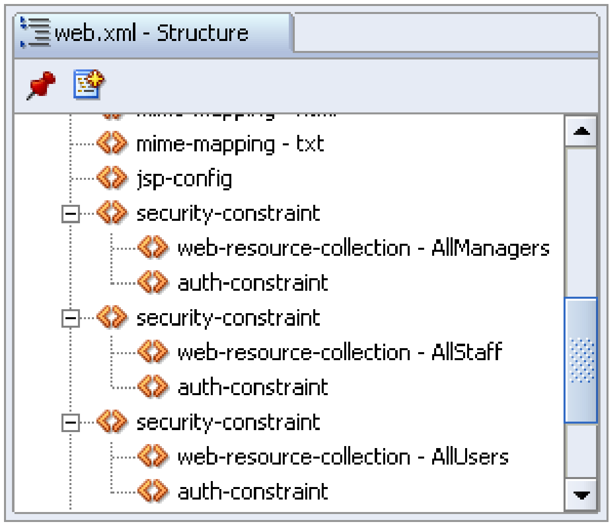
Now that you have defined the security roles, they need to be assigned to security constraints to enforce the authorization.

4. Select the **Security Constraints** node and click **New.** On the Web Resources tab, click **Add** and specify **`AllManagers`** as the Web Resource Name. Click **OK**.
5. With the **`AllManagers`** collection selected, click the **Add** button in the lower pane on the right side of the URL Patterns tab. In the Create URL Pattern pop-up window, enter **`faces/app/management/*`**. This enables all managers to access any files in the management directory.
6. With the **`AllManagers`** collection selected, click the **Authorization** tab and select the **manager** check box.
7. Create two more security constraints using the values in the following table:

Web Resource Collections	URL Patterns	Authorization
AllStaff	`faces/app/staff/*`	`technician` and `manager`
AllUsers	`faces/app/*`	`user`, `technician`, and `manager`

8. Click **OK** to close the Web Application Deployment Descriptor dialog box.

   When you finish this step, there should be three constraint entries under the Security Constraints node in the structure window. Each constraint entry corresponds to one of the collections from the preceding table. With the `web.xml` file open in the editor, the Structure window should show all the constraints, as in the following screenshot:



9. With the SRList page open in the Visual Editor, right-click and select **Run**. When prompted, enter `bernst` as the username and `welcome` as the password. You should see service requests appear in the list. Select some of the menu options for Open, Pending, and Closed requests.

Summary

In this chapter, you provided security for the SRDemo application. To accomplish this, you performed the following key tasks:

- Created a container to control security
- Created a class to manage roles
- Created a class to provide authentication
- Integrated a user with the application
- Exposed the user e-mail ID to the application
- Set up container security
- Set up application access

7

Developing a Search Page

This chapter describes how to build the Search page using JavaServer Faces and ADF components. The page contains two sections: one to specify the query criteria and the other to display the results. You create buttons enabling the user to select a record and to view or edit the record.

The chapter contains the following sections:

- Introduction
- Creating the Search Page
- Adding Data Components to the Search Page
- Modifying the Default Behavior of a Query
- Wiring Up the Edit Button
- Wiring Up the View Button
- Summary

Introduction

The SRSearch page provides a query-by-example screen for technicians and managers to search the entire list of service requests. The page is divided into two areas: a query area at the top (which is always in query mode) and a results area in the form of a table (which displays the results of the last search).

Note: If you did not successfully complete Chapter 6, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter7` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter7`.
2. Unzip `<tutorial_setup>\starterApplications\SRDemo-EndOfChapter6.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter7\SRDemo\SRDemo.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 6.

Use the page-flow diagram to see how the search page fits in the application. Here are some points to note about the page:

- The returned records appear on the same page as the query area.
- The page is always in query mode.
- The returned records can be selected and then viewed or edited.
- Only managers can view all records; others can view only their own records.

Creating the Search Page

The first task in creating the Search page is to build its structure using ADF Components and then adding the data component from the data model.

Note: Earlier in the tutorial, you created the page outline in the page-flow diagram. Now you complete the page and apply the template that you created in Chapter 4. You could also create the page from the New Gallery by using the JSF JSP item.

Perform the following steps to create the SRSearch page and attach the template that you created earlier:

1. If it is not open, double-click the `faces-config.xml` file to view the page-flow diagram.

2. Double-click the **SRSearch** page to invoke the JSF Page Wizard.

3. Complete the wizard using the following values:

Wizard Step 1: JSP File

Field	Value
File Name	`SRSearch.jspx`
Directory Name	This is the location where the file is to be stored. Append **`\app\staff`** to the default values, placing the file in its own subdirectory. The directory should be `<jdev_install>\jdev\mywork\SRDemo\UserInterface\public_html\app\staff`.
Type	`JSP Document`
Mobile	Clear the check box.

4. Click Next to continue.

Wizard Step 2: Component Binding

Field	Value
Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.
Name	`backing_app_staff_SRSearch`
Class	`SRSearch`
Package	`oracle.srdemo.userinterface.backing.app.staff`

5. Click Next to continue.

Wizard Step 3: Tag Libraries

Field	Value
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

6. Click **Finish** to create the page details. The new SRSearch page is displayed in the Visual Editor.

7. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, right-click the **afh:html** node and choose **Copy** from the shortcut menu.

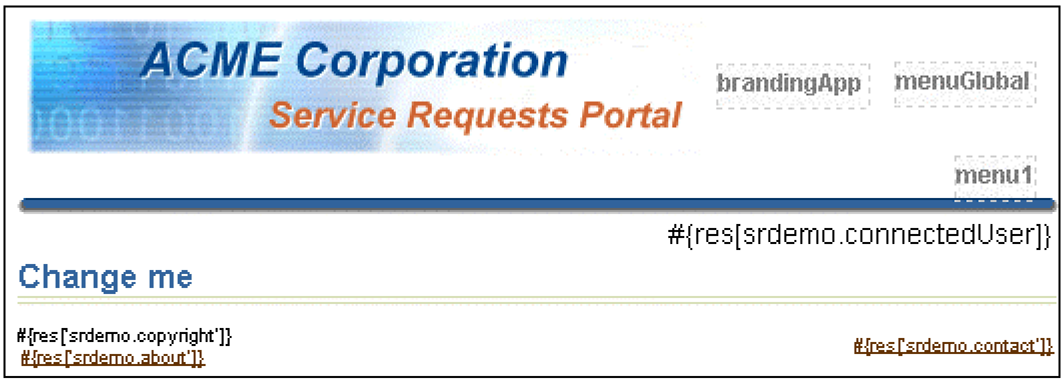
8. Click the tab to return to the SRSearch page. In the Structure window, expand the **f:view** node.

9. Delete the **html** node. Then right-click **f:view** and choose **Paste** from the shortcut menu.

10. The main image doesn't appear; you need to change the path. Select the image and then, in the Property Inspector, reset the URL property to `/images/SRBranding`.

11. Double-click **afh:head** and set the Title property to `SRDemo Search`. The value of this property becomes the browser title.

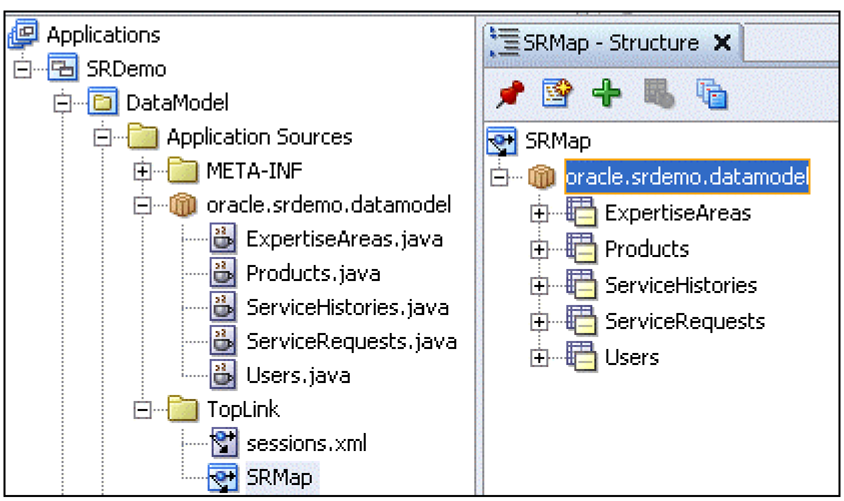
12. The Visual Editor now displays the SRSearch page with the look and feel of the other pages:



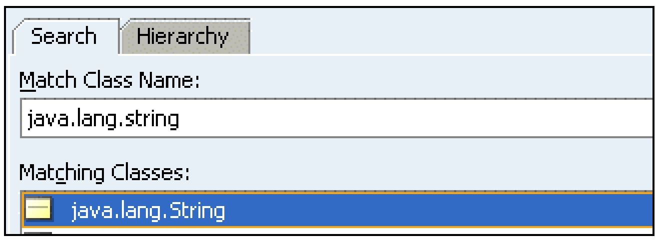
Creating a Named Query

The search form uses a named query that is based on ServiceRequests and accepts two parameters.

1. In the Applications Navigator, select the **SRMap** node in the DataModel project. The structure of the SRMap is displayed in the Structure window:



2. Double-click the **ServiceRequests** node, and then click the **Queries** tab. Click the **Add** button to create a new named query.
3. In the Add TopLink Named Query pane, type **`searchServiceRequests`** as the name of the new query. Click **OK** to continue.
4. With the `searchServicesRequests` named query selected, click the **General** tab. In the Parameters area of the editor, click the **Add** button.
5. In the Class Browser window, click the **Search** tab. This is where you define the parameter type.
6. Enter **`java.lang.String`** as the Match Class Name property. As you type, the list becomes more refined. When the `java.lang.String` matching class is highlighted, click **OK** to define the type.

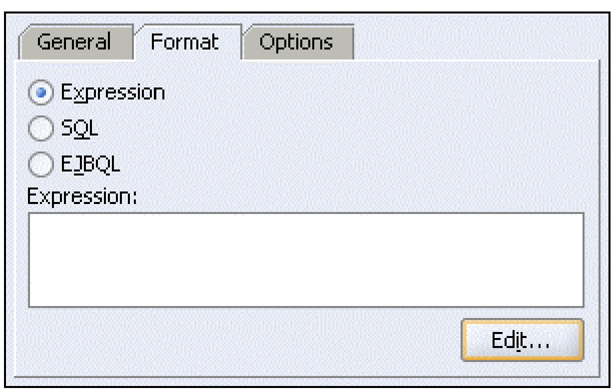


7. In the Parameters area, change the parameter name to **`descr`**.
8. Create a second parameter of `java.lang.String` and name it **`status`**.

Creating the Query Expression

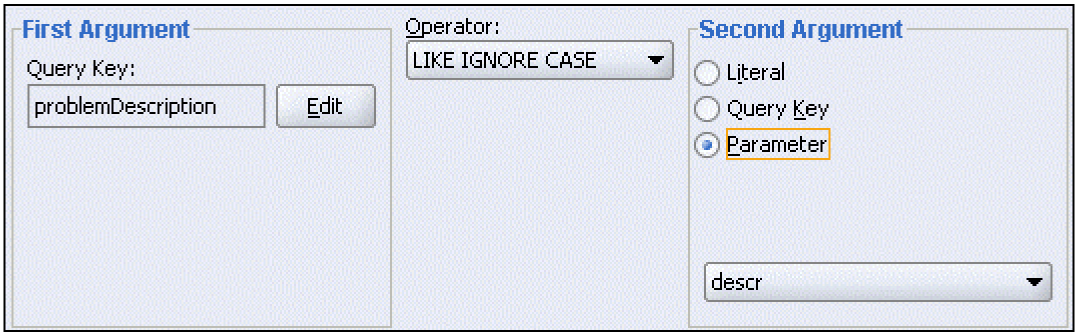
Now that you have defined two parameters (`descr` and `status`), you need to define an expression using them. This expression defines how the parameter is associated with an attribute in the named query. The expression is evaluated at run time to determine the rows returned by the named query.

1. With the **searchServiceRequests** named query selected, click the **Format** tab and then click the **Edit** button in the Parameters area of the editor.



2. In the Expression Builder, click **Add** to create a new expression.
3. In the First Argument area of the expression, click **Edit**. In the Choose Query Key pane, select **`problemDescription`**.
4. Click **OK** to continue.

5. Back in the Expression Builder, set the Operator to **`LIKE IGNORE CASE`**, and set the Second Argument to the **`descr`** parameter that you created earlier. It should look like the following screenshot. Click **OK** to continue.

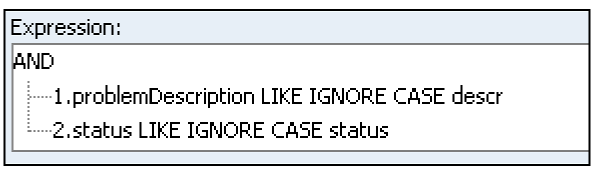


6. Next, add a second expression. Click the **Add** button to add a new expression. Set the expression component to the values in the following table:

Field	First Argument	Operator	Second Argument (parameter)
Second Component	`status`	`LIKE IGNORE CASE`	`status`

When complete, your expression should look like the following. Confirm your results, make changes if necessary, and click **OK** when done.

7. Save your work

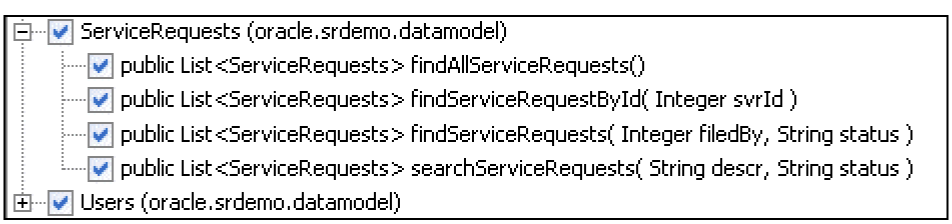


Re-creating the DataControl

The last part of creating this named query is adding it to the data controls. Because you have added code to some of the session bean methods, you need to make sure you don't override them as you re-create the data control.

1. Right-click **SRPublicFacadeBean.java** in the Applications Navigator and select **Edit Session Facade** from the context menu.

2. Expand **ServiceRequests** and make sure the new named query **searchServiceRequests** is selected.



3. Click **OK**

4. Right-click **SRPublicFacadeBean.java** in the Applications Navigator and select **Create Data Control** from the context menu.

5. In the Data Control pane, you now see `searchServiceRequests(String, String)`.

Adding Data Components to the Search Page

In this section, you modify the SRSearch page to display a different title and include data-bound controls. In most cases where you define titles and prompts, you reference the values defined in the data model rather than hard-coding them in the page. As a result, if the model's prompts or titles change, then these updated values are used at run time.

1. Click the **SRSearch.jspx** tab and, in the Structure window, expand the **f:view→afh.html→afh.body→h:form** nodes to expose `af:panelPage`.

2. Double-click **`af:panelPage`** and, in the properties, set the Title field to **`#{res['srsearch.pageTitle']}`**.

   This value is set in the template, so you could refer to the `res` variable when the template is first used or on each page that uses the template. You could set this property to the literal text you want for the title. We are setting it to a value in the `UIResources.properties` file, which contains a list of paired properties and values. At run time, this file is read and the values are replaced on the page. Throughout the tutorial, you use this convention for button names, page titles, and field prompts. At run time, the value should be "Find a Service Request."

3. If you click the **Source** tab in the Editor, you see the code. The following screenshot shows the source code for this step:

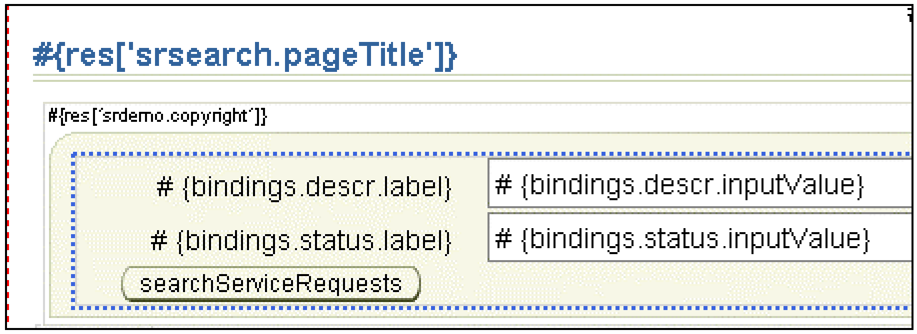
```
<af:panelPage title="#{res['srsearch.pageTitle']}"
              binding="#{backing_app_staff_SRSearch.panelPage1}"
              id="panelPage1">
```

4. Click the **Design** tab to see the changes.

5. In the component palette, open the ADF Faces Core page and drag a **PanelBox** to the **`af:panelPage`** in the structure window. This action creates a placeholder for the query component of the page.

You now add a Parameter Form that uses the searchServiceRequests named query. On the Data Controls tab, expand the **`SRPublicFacadeLocal`** control, and locate **searchServiceRequests(String, String)**.

6. Select **searchServiceRequests(String, String)** and drag it to `af:panelBox`. Select **Parameters| ADF Parameter Form** from the context menu. Click **OK** in the Edit Form Fields dialog box to accept the defaults. These actions add a form with two fields and a search button.



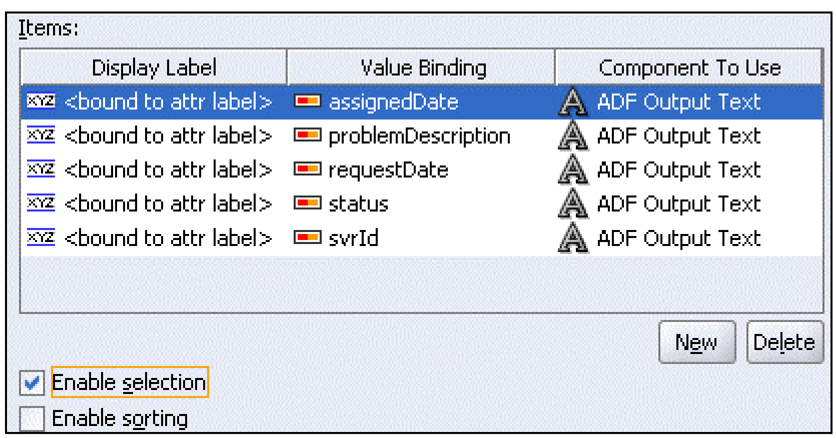
7. Change the searchServiceRequests button's Text property to use the resource **`#{res['srsearch.searchLabel']}`**.
8. Change the Label property of the description `af:inputText` component to '**Description:**'.
9. Change the Label property of the status `af:inputText` component to '**Status:**'.

Adding Data Components

In the next few steps, you create the area and data components to display the results of the query:

1. In the Data Control palette, expand **searchServiceRequests** and drag the subnode **ServiceRequests** to the **`af:panelPage`**.
2. In the pop-up menu, select **Tables | ADF Read-only Table**.

   This table displays the search results. You see the Edit Table Columns window, where you can change the display label, the binding value, and the type of component used to display it. The default values originate from those defined in the data model.
3. Select the **Enable selection** check box and accept the defaults. Click **OK** to continue..



4. In the Structure window, expand the **af:table** node and reorder the columns of the table to **srvId**, **problemDescription**, **status**, **requestDate**, and **assignedDate**.

As you select the node in the Structure window, the Visual Editor displays the column so that you can view the completed column definition.

5. In the Structure window, expand the **af:table → Table facets → selection**, and double-click the **af:tableSelectOne** node. Change the hard-coded default Text property to the resource string **`#{res['srsearch.resultsTable.prefix']}`**. This action changes the column's label to Results as defined in the `UIResources.properties` file.
6. With the af:table selected in the structure window, change the **Id** property to **`srtable`**.
7. Delete the **Submit** button.
8. **Save** the page.
9. From the ADF Faces Core drop-down menu of the Component palette, drag two **ADF Faces Core → CommandButtons** into the `af:tableSelectOne` node.
10. Double-click each **af:commandButton** and change the Text property to the values in the following table.

    This action associates each button with one of the two functions you want it to carry out: edit or view. Both of the values in the following table are defined in the `UIResources.properties` file.

Command Button	Value
1	`#{res['srsearch.resultsTable.edit']}`
2	`#{res['srsearch.resultsTable.view']}`

Modifying the Default Behavior of the Query

The default behavior of the parameter form is to simply accept the values the user enters and execute the query. Which means the user must put a value or a wildcard in each of the fields for the query to return results. That is probably fine if it is a two-field form, but more than that would present usability issues to users.

In this section, you add some Expression Language (EL) to the page-definition parameters that will insert a wildcard if the user doesn't enter a value in the field. You add this code to each of the parameters so that the can enter either or both parameters.

1. Select the page definition for the **SRSearch** page in the Applications Navigator.
2. In the Structure window, expand **bindings → searchServiceRequests**.
3. Double-click the **descr** property.
4. For the NDValue in the NamedData Properties dialog box, click the **(…)** button (browse/edit).

    The expression is what populates the parameter before it is passed to the query for execution. By default, the parameter is populated from the related field on the parameter form. The code you are about to add checks for a value and returns a wildcard ("%") if the parameter is either null or blank.

5. Replace the default expression for descr with the following code:

```
${((bindings.searchServiceRequests_descr == null) ||
(bindings.searchServiceRequests_descr == '')) ? '%' :
bindings.searchServiceRequests_descr }
```

The preceding code tests `descr` for null and for blank (''). If either is true, the expression returns a wildard ('%'). If `descr` is not null, the expression returns the value that the user entered.

6. Click **OK** to accept the changes. Click **OK** again to close the dialog box.
7. Double-click the status property
8. For the **NDValue** in the Named Data Properties dialog box, click the **(...)** button (browse/edit).
9. Replace the default expression for status with the following code:

```
${((bindings.searchServiceRequests_status == null) ||
(bindings.searchServiceRequests_status == '')) ? '%' :
bindings.searchServiceRequests_status }
```

10. Right-click anywhere on the page and select **Run**. When prompted, enter **`sking`** as the username and **`welcome`** as the password.
11. Click the search button without entering values in either of the fields. The result should show all the service requests.
12. Enter **`open`** in the status field and click search. The result should be all the Open service requests.
13. Enter a value in the description field (for example, **`"%wash%"`**), and then click search. You should now see the rows with "`wash`" somewhere in the description with a status of **open**.
14. Clear the **status** field and click search. You should now see all the rows with "`wash`" somewhere in the description.

Adding a Refresh Condition

When you first ran the page, the query was executed when the page was loaded. Because you added the code to substitute wildcards for null values, the initial query returns and displays all rows.

In the final few steps, you add a refresh condition that keeps the form from executing the query until the user clicks the search button.

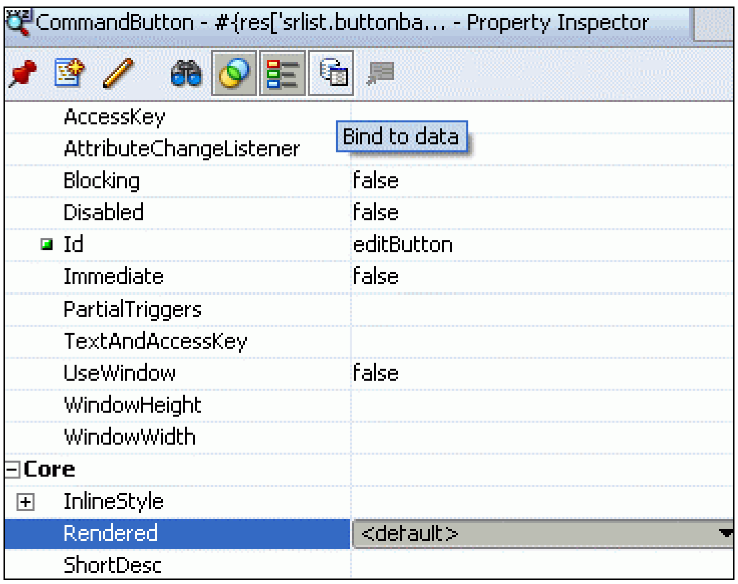
1. Select the page definition for the **SRSearch** page in the Applications Navigator.
2. In the Structure pane, expand **executables**.
3. Click **searchServiceRequestsIter**.
4. In the Properties Inspector, change the RefreshCondition to **`${adfFacesContext.postback}`**.
5. Right-click anywhere on the page and select **Run**.
6. Notice that the detail table does not display any rows until you click search.

Wiring Up the Edit Button

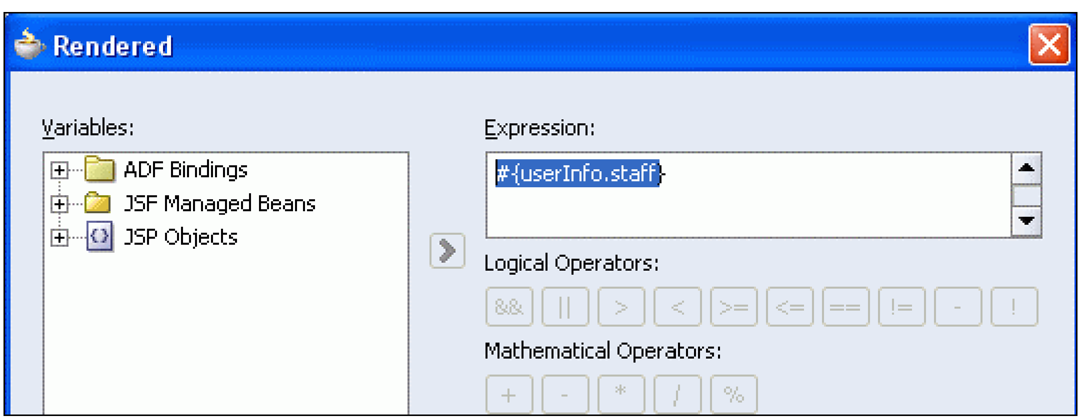
The next two steps are the same as those when you created the SRList page. The Edit button is available only when a technician or manager logs in. Clicking the button takes users to the SREdit screen, where service request details can be modified.

Perform the following steps to specify the Edit button functionality:

1. Click the **edit** button. In the Property Inspector, set the Id property to **editButton**.
2. Specify that you want to display the Edit button only when the user who is currently logged in is a member of staff. Select the **Rendered** property, and click the **Bind to Data** button in the toolbar of the Property Inspector (it is the second button from the right; use the tooltips labels to check which button you need).



3. In the Rendered dialog box, type **`#{userInfo.staff}`** in the Expression field. Click **OK**. This is to verify that the logged-in user is indeed a member of staff.



Adding Backing Bean Code

Create a method in the backing bean that passes the ID of the currently selected service request through to the Edit page, so that the appropriate record can be retrieved and displayed.

1. In the Visual Editor, double-click the **Edit** button to invoke the backing bean.
2. In the Bind Action Property dialog box, click **OK** to add the `editButton_action` method to the backing bean. In the backing bean file, replace the generated code in the `editButtonAction` method with the following code to specify its navigation path.

   ```
   setCurrentSvrIdFromRow();
   return "edit";
   ```

3. The `setCurrentSvrIdFromRow` method does not yet exist. Click the **CodeAssist** icon (the light bulb in the left margin) to create it.
4. Implement the `setCurrentSvrIdFromRow` method by adding the following code:

   ```
   FacesContext ctx = FacesContext.getCurrentInstance();
   JUCtrlValueBindingRef tableRowRef =
    (JUCtrlValueBindingRef)this.getSrtable().getRowData();

   Integer svrId =
   (Integer)tableRowRef.getRow().getAttribute("svrId");

   UserSystemState.storeCurrentSvrID(svrId);

   //Store away where we want to come back to
   UserSystemState.storeReturnNavigationRule
   ("GlobalHome");
   ```

   This code does two things:

   - It extracts the service request ID and stores it in the `UserState` managed bean.
   - It sets the navigation path to return to the SRList page on completion of the edit.

5. Still in the backing bean, add a new class variable as follows:

   ```
   private BindingContainer bindings;
   ```

   (Press [Alt] +[Enter] to import the package `oracle.binding` when prompted by Code Assist.)

6. Right-click **`bindings`**, and select **Generate Accessors** from the context menu. In the Generate Accessors menu, click **OK** to generate both the `setBindings` and the `getBindings` methods.
7. Save the **`SRSearch.java`** file.

Wiring Up the View Button

The View button is available to all users of the application, enabling them to navigate to the SRMain page to update the history of a selected request.

Perform the following steps to specify the View button functionality:

1. Return to `SRSearch.jspx` in the Visual Editor and select the second Command Button (the **view** button).
2. In the Property Inspector, set the Id property to **`viewButton`**.
3. As with the Edit button, you need to create a method in the backing bean that specifies exactly what must happen when the View button is clicked. Double-click the **View** button to invoke the backing bean for the page. In the Bind Action Property dialog box, clibernstck **OK** to add the **`viewButton_action`** method to the backing bean.
4. In the `SRList.java` file, replace the generated code in the `viewButton_action()` method with the following code:

   ```
   return drillDown_action();
   ```

5. As before, click the **Code Assist** button to create the `drillDown_action` method.

   This method again uses the `setCurrentSvrIdFromRow` method that you created in the previous section. It passes the ID of the currently selected service request through to the SRMain page, so that the appropriate record can be retrieved and displayed.

6. Add the following code below the method to specify this behavior:

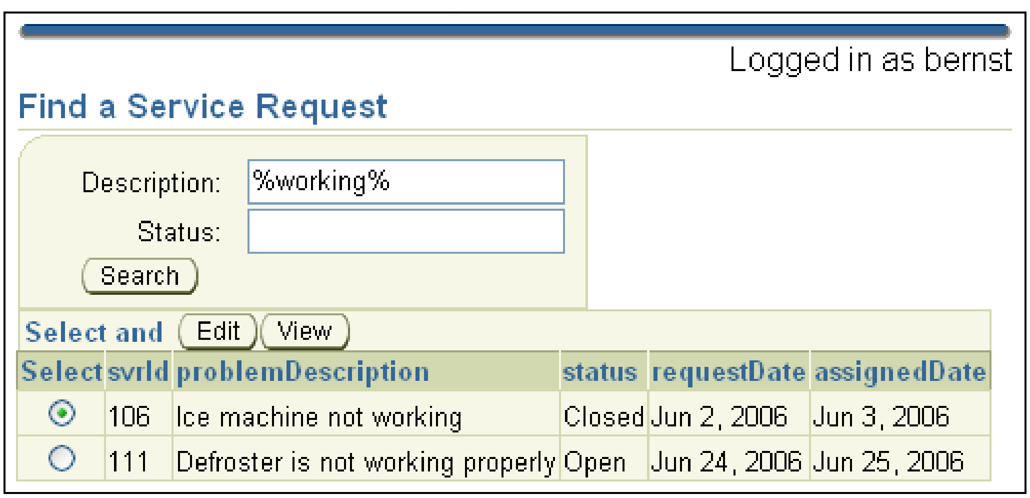
   ```
   setCurrentSvrIdFromRow();
   return "view";
   ```

7. Save the file.

   Now that the page is completed, you can run it and examine the functionality.

8. With the SRSearch page open in the Visual Editor, right-click and select **Run**. When prompted, enter **`bernst`** as the username and **`welcome`** as the password. Test the search function. (**Note:** You are unable to use the View and Edit buttons until you complete the SRMain and SREdit pages in later chapters).

The page should look something like the following.



Summary

In this chapter, you built the Search page using JavaServer Faces and ADF components. To accomplish this, you performed the following key tasks:

- Created the Search page
- Added data components to the Search page
- Wired up the View and Edit buttons
- Modified the default behavior of a query

8

Developing a Master-Detail Page

In this chapter, you develop a master-detail page that shows a service request and its Service Request History rows. From this page, users can see the scope and history of a service request. They can also add detailed notes to the service request.

The chapter contains the following sections:

Introduction

The SRMain page provides a master-detail view of service requests and their Service Request History rows. The page contains three component areas: the read-only Service Request form, the Notes input area, and the Service Request History table.

You perform the following key tasks in this chapter:

- Create the Service Request read-only form
- Create the Notes input form and add code to programmatically set data values from parameters
- Add the Service Request History table

Note: If you did not successfully complete Chapter 7, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named **`Chapter8`** to hold the starter application. If you used the default settings, it should be in **`<jdev_install>\jdev\mywork\Chapter8`**.
2. Unzip **`<tutorial_setup>\starterApplications\SRDemo-EndOfChapter7.zip`** into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select **`<jdev_install>\jdev\mywork\Chapter8\SRDemo\SRDemo.jws`**. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 7.

Developing the Basic UI

In the first part of this chapter, you create the SRMain page and add the basic UI components. These are the layout components that you will use to hold the data-aware components that you add later.

Perform the following steps to create the SRMain page and copy the template you created earlier into this page:

1. If it is not open, double-click the **`faces-config.xml`** file to view the page-flow diagram.
2. Double-click the **/app/SRMain.jspx** page to invoke the JSF Page Wizard.
3. Ensure that the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File	
Field	**Value**
File Name	`SRMain.jspx`
Directory Name	The directory name should be "...`public_html\app`."
Type	JSP Document
Mobile	Clear the check box.

4. Click **Next** to continue.

Wizard Step 2: Component Binding	
Field	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.
Name	`backing_app_SRMain`
Class	`SRMain`
Package	`oracle.srdemo.userinterface.backing.app`

5. Click **Next** to continue.

Wizard Step 3: Tag Libraries	
Field	**Value**
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

6. Click **Finish** to create the page details. The new SRMain page is displayed in the Visual Editor.
7. Open **SRDemoTemplate.jspx** in the Visual Editor (if it is not already open). In the Structure window, shrink the **afh:html** node and select it. From the shortcut menu, choose **Copy**.
8. Click the tab to return to the SRMain page, and in the Structure window select **`f:view`** node.
7. Delete the **`html`** node. Then right-click **`f:view`** and choose **Paste** from the shortcut menu. The look and feel that you created earlier is now applied to the new page.

Your page should now look like the following:



Adding Service Request Components

Now that you have created the basic page, you can begin to add data-aware components that display the service request data.

You first need to add some ADF Layout components that help with the alignment and format of the page. You now add an ADF tableLayout component with three rowLayout components; each of the rowLayout components will hold a section of this page.

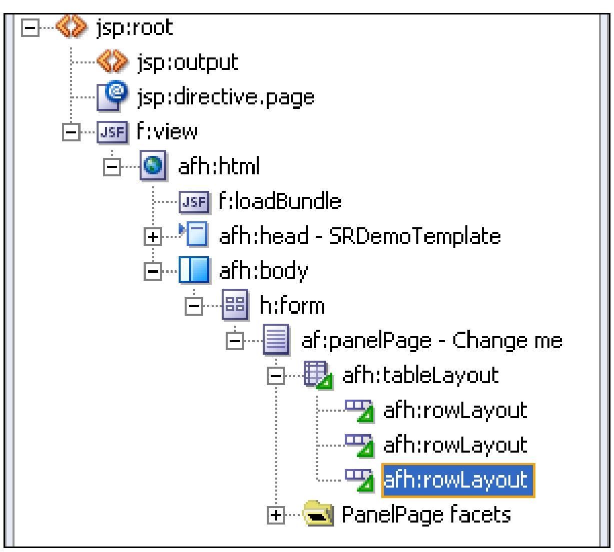
1. Select an **ADF Faces HTML → TableLayout** component and drag it to **`af:panelPage`** in the Structure pane.

   This component is like an HTML table except that it can be manipulated programmatically using the backing bean. For example, you can set the `Rendered` property to false, which disables the display of this table and all of its components.

2. Add an **ADF Faces HTML → RowLayout** component and drag it to the **`afh:tableLayout`** component that you just added. This component provides a layout object and (like the tableLayout component) can be changed programmatically.

3. Repeat the previous step to add a second and third **rowLayout** component.

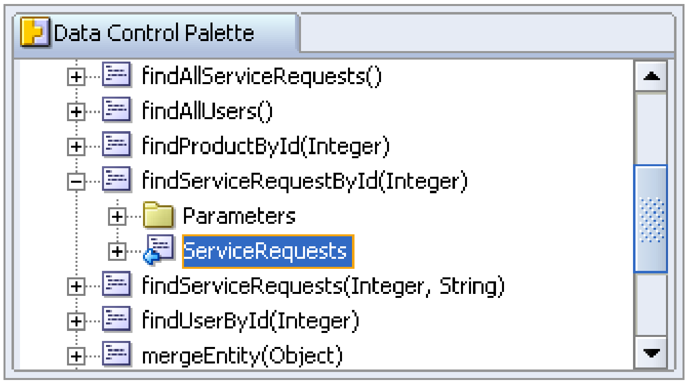
   The Structure window should now look like the following screenshot:



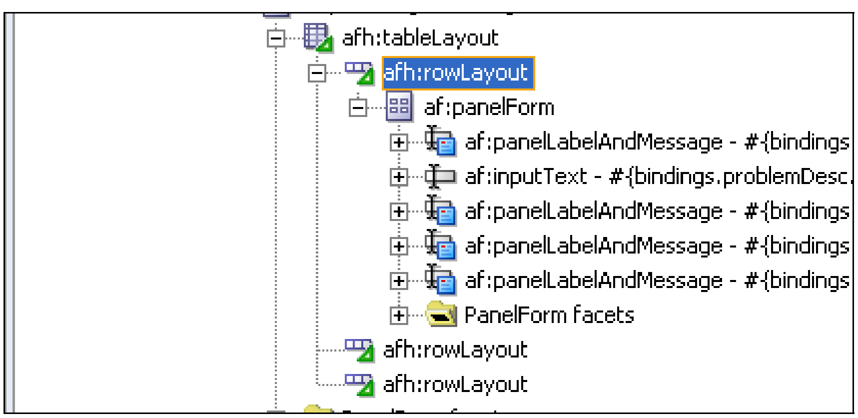
Adding Data-aware Components

Perform the following steps to add an ADF data-aware component to display service requests:

1. Click the **SRMain.jspx** tab in the Visual Editor to open the page again.
2. Select the **Data Control palette** and expand the **SRPublicFacadeLocal** node.
3. Drag the result collection of **findServiceRequestById(Integer)** to the first `afh:rowLayout` in the Structure window as an **ADF Read-only Form**. The collection is the node under the method name.



4. In the Edit Form Fields dialog box, change the **problemDescription** column to use an ADF Input Text w/Label component. Click in the **Component To Use** column next to problemDescription and select **ADF Input Text w/Label**. This changes the problemDescription field to a browser input text widget. You can then change display properties such as height and width. Click **OK**.
5. In the Action Binding Editor, enter a default value for the findSvrID parameter. The Value property determines the source of the parameter. Use the EL browser **(...)** to pick **`${userState.currentSvrId}`** from the JSF Managed Beans node. Click **OK**
6. Check the Structure window to make sure it looks like the following:



Refining the Layout

Perform the following steps to use the Structure window to add and arrange components. This is an easy way to ensure that your layout components are in the correct locations relative to the other components.

1. Rearrange the **af:panelLabelAndMessage** components so that the **svrID** component is first and **problemDescription** is last. You can do this by dragging the components to the correct relative positions using the Structure window.

2. Change the title of the panelPage from Change Me to **Service Request Information**. (Select **af:panelPage** in the Structure pane and change the Title property in the Properties Inspector.)

3. Using the Structure window and the Properties Inspector again, change the number of rows the problemDescription displays. Select the **problemDescription** and set the Rows property to **4**.

   Because problemDescription is a multiline field, this enables more of the description to be displayed. Changing the Rows property also automatically adds scrolling capability to the widget.

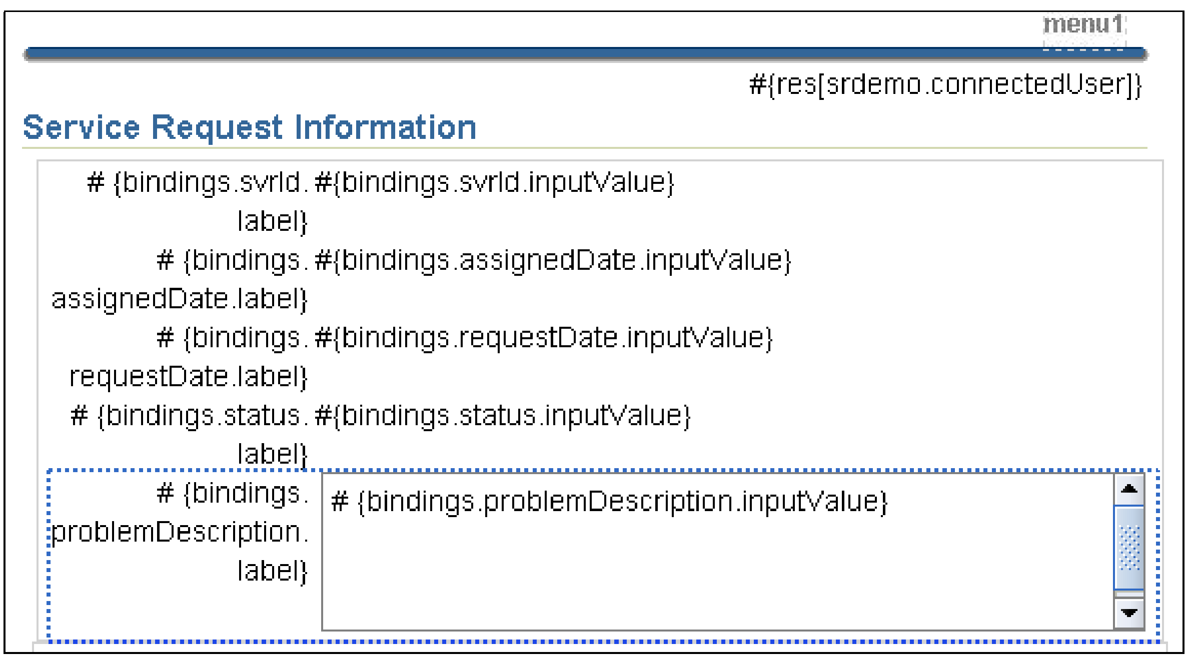
4. Change the **Columns** property to **35**.

   This makes the field 35 characters wide. You could leave this property blank and enable JSF to use a default value, but this gives you finer grain control of the layout.

5. Change the **ReadOnly** property to **true**.

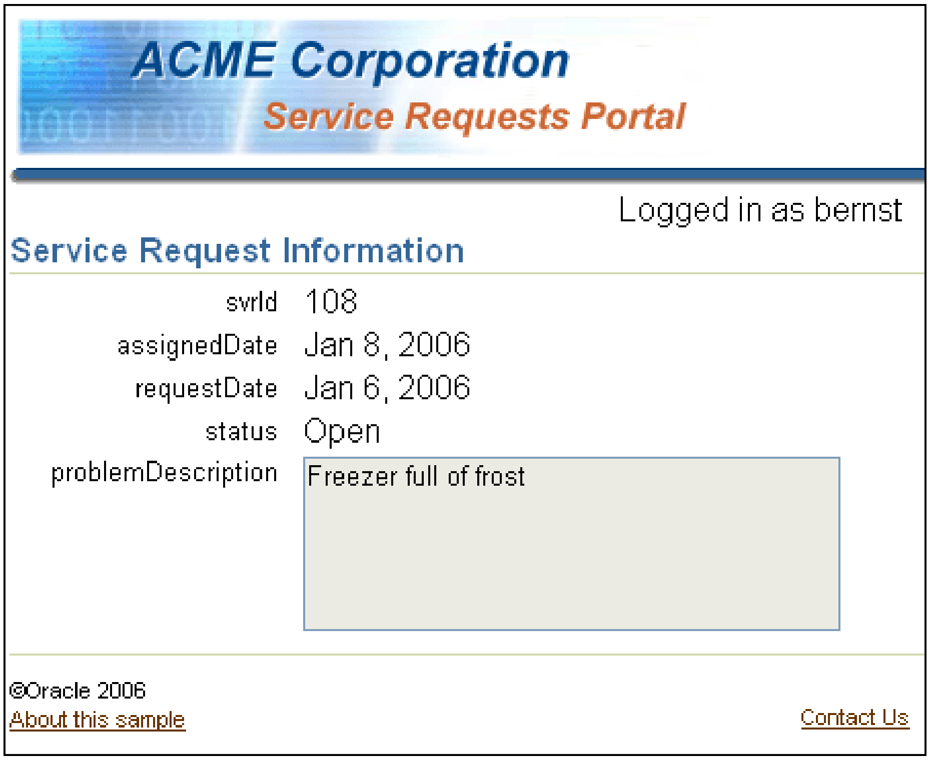
   The default for an input text component is to be updateable, but because this portion of the form is read-only, you should change this field to readOnly.

The page should look like the following screenshot:



6. The SRMain page gets its service request context from userState.currentSvrId. Although you can run this page as a stand-alone, it won't display data unless it is called from the SRList page. To run the page and see the results so far, right-click the **SRList** page and select **Run** from the context menu.

   Sign on to the application using `bernst` as the username and `welcome` as the password. The SRList page displays all of the service requests for the current user. Select any service request and click the **View** button. You now see your SRMain page with the selected service request. The page should look something like the following:



Adding the Notes Panel

The second section of this page is an entry form that enables users to add a new line to the ServiceRequest History. The form displays only the notes field and a button.

The notes panel is based on a serviceHistories custom constructor. This constructor is called when the page is loaded. This creates a row in the iterator that accepts values from the form and is also accessible using EL. The constructor accepts two arguments: the current service request and the current user object.

Create the custom constructor by following these steps.

1. Expand **DataModel → Application Sources → oracle.srdemo.datamodel** in the Applications Navigator.

2. Double-click **ServiceHistories.java** to open it in the code editor.

3. Add the following code just below the default constructor. The default constructor is defined as

```
public ServiceHistories() {
```

```
    …
    }

    public ServiceHistories(ServiceRequests sr, Users user) {
        this();
        sr.addServiceHistories(this);
        setUsers(user);
        setLineNo(null);
        setSvhDate(null);
    }
```

4. After you create the constructor, re-create the data control. Right-click **SRPublicFacadeBean.java** and select **Create Data Control** from the context menu.

Creating the Add Note Section

Now that you have the custom constructor created and included in the data control, you can add the form to create service history rows. When you add the form, you are prompted for the two arguments to the constructor. The first is the current service request, which you obtain from the findServiceRequestById binding. The second is the user object, which you obtain from the userInfo bean.

1. Click the **SRMain.jspx** tab in the visual editor.
2. Select the Data Control Palette and drag the **SRPublicFacadeLocal → Constructors → oracle.srdemo.datamodel.ServiceHistories → ServiceHistories(ServiceRequests, Users)** method to the second **`afh:rowLayout`** in the Structure Window.
3. In the Edit Form Fields dialog box, select **Include Submit Button** then click **OK** to accept the default field values.
4. In the Action Binding Editor, select the **sr** parameter and double-click the **value** property. Click the **browse** button to open the EL picker.
5. Expand **ADF Bindings → bindings → findServiceRequestByIdIter → currentRow.** Double-click **dataProvider** to create the expression. Click **OK**. The code should look like the following:

   `${bindings.findServiceRequestByIdIter.currentRow.dataProvider}`
6. Double-click the value property for the **user** parameter and click the browse button to open the EL picker.
7. Select **JSF Managed Beans → userInfo** and double-click **userobject** to add it to the expression. Click **OK**. The code should look like the following:

   `${userInfo.userobject}`
8. Click **OK** to close the Action Binding Editor.

   When users add a note using this panel, the only field they need to use is the Notes field. In the next few steps, you will change the rendered property of all the fields except the Notes field to false using the Structure pane and the Properties Inspector.
9. Select **af:**inputText items in the structure pane and change the **rendered** property to **false** for each attribute *except* notes.inputValue.

   Note: You can press [Ctrl] + [Click] to multiselect all of the attributes and change the common rendered property in the property inspector.

10. Change the **Rows** property of the **notes.inputValue** component to **4**. This makes the field into a text area type field.

11. Change the Columns property to **35**.

12. Change the Label property to **SR Notes**.

Deriving Values in Code

The last thing you need to do to this panel is add custom code to an "Add a Note" button that adds data to and persists the history row. Instead of having the user enter each of the values, you derive some of the values from the current service request and the current user.

1. In the Data Control Palette, select **SRPublicFacadeLocal → persistEntity(Object)** and drag it to the Submit button in the visual editor. Select **Bind Existing CommandButton** from the context menu.

2. In the Action Binding Editor, set the value property of the entity parameter to **${bindings.ServiceHistories.result}**. Click **OK** to continue.

3. Select the **persistEntity** button and change the following properties.

Field	Value
Text	`Add a Note`
Id	`addNoteButton`

We want to be able to refresh some data a little later on, but only when this button is clicked. ADF provides a setActionListener component that is executed only when the button is clicked. The setActionListener accepts two arguments: a "from" clause and a "to" clause. In a refresh condition, we will check the value of the "to" clause to see if the button has been pressed.

4. Right-click **af:commandButton – Add a Note** in the Structure window. Select **Insert Inside af:command Button – Add a Note → ADF Faces Core → SetActionListener** from the context menu.

5. Enter the following values in the Insert SetActionListener dialog box, and then click **OK**.

Field	Value
From	`#{true}`
To	`#{requestScope.createNewSH}`

6. Double-click the **Add a Note** button. This opens the Bind Action Property dialog box. Make sure the **Generate ADF Binding Code** check box is selected and then click **OK** to accept the default value for the method name.

   The Bind Action Property dialog box closes and directs you to the `addNoteButton_action()` method in the `SRMain.java` file. This is where you add the custom code to set the svhType argument, persist the entity, and requery the service

request.

The following code is executed when the user clicks the “Add a Note” button. Remember that the user is only entering the notes description on the form. The code determines the type of the note based on the user. If the signed-on user is a customer, the value is set to Customer; if the user is staff, the type is set to Technician.

7. Determine the svhType as follows: Copy the code below to the `AddNoteButton_action()` method as the first lines in the method.

```
// START CUSTOM CODE TO SET SVHTYPE

FacesContext ctx = FacesContext.getCurrentInstance();
String callType = "Customer";
UserInfo user =
    (UserInfo)JSFUtils.getManagedBeanValue(ctx, "userInfo");
    if (user.isStaff()) {
        callType = "Technician";
    }
ADFUtils.setPageBoundAttributeValue(getBindings(),"svhType",
callType);
```

8. When you copy this code into the method, JDeveloper prompts you to import the required classes. Accept the following imports:

```
javax.faces.context.FacesContext

oracle.srdemo.view.UserInfo

oracle.srdemo.view.util.JSFUtils

oracle.srdemo.view.util.ADFUtils
```

9. Persist the entity as follows: JDeveloper created the persist entity code for you when you dropped the `persistEntity()` method on the submit button. This is the code you want to execute next. It is already in the `addNoteButton_action()` method, so no changes are required.

```
BindingContainer bindings = getBindings();

OperationBinding operationBinding =
    bindings.getOperationBinding("persistEntity");

Object result = operationBinding.execute();
```

10. Refresh the service request iterator as follows: When the user adds a note, and you persist the entity, the table at the bottom of the page (which you add in the next section) needs to be refreshed. You refresh the iterator by getting the OperationBinding for the method and executing it. Add the following code after the persist entity code.

...

```
Object result = operationBinding.execute();

//now re-execute the iterator to refresh the screen

OperationBinding requery =
   bindings.getOperationBinding("findServiceRequestById");

requery.execute();
```

That’s all of the code for the addNoteButton_action() method.

There is one last thing you need to address, which is to specify when and how often the `ServiceHistoriesIterator` is automatically refreshed.

11. Select **app_SRMainPageDef.xml** in the Applications navigator.
12. Select **ServiceHistoriesIter** in the Structure window.
13. In the Properties Inspector, change the RefreshCondition to the following:

```
${(!adfFacesContext.postback || requestScope.createNewSH) and
          empty bindings.exceptionsList}
```

This condition checks to make sure that the automatic refresh happens if the page refresh is not because of a Faces postback or if the createNewSH is TRUE. Recall that this is the code you added in the setActionListener event. It also prevents a refresh if there are binding errors in the exceptions list.

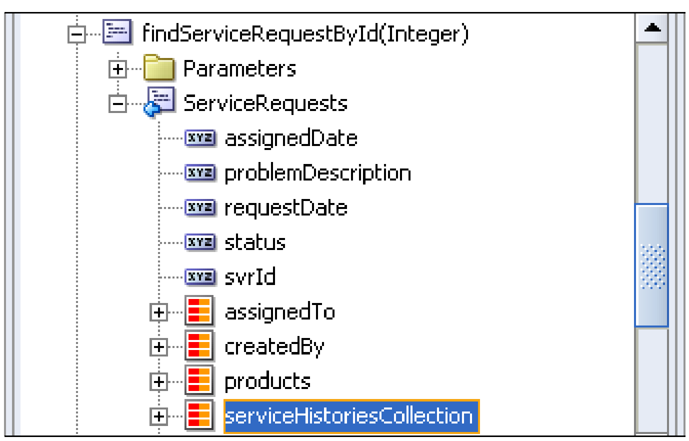
Because you are now committing records to the database, you need to set the OC4J preferences to manage the transactions as local transactions. You do this in the OC4J preferences.

14. Make sure the OC4J server in not running. Select **Run > Terminate OC4J** from the menu.
15. Click the **DataModel** project in the Applications Navigator, and then select **Tools > Embedded OC4J Preferences** from the menu.
16. Expand **Current Workspace (SRDemo) → DataSources**.
17. Click **jdev-connection-managed-SRDemo <Managed Data Sources>**.
18. Set Transaction Level to **Local** and click **OK**

Adding the Service Histories Panel

The final UI component that you need on the SRMain page is a read-only panel that displays the Service Request History records for the current service request.

1. Click the **SRMain.jspx** tab in the visual editor.
2. Expand **findServiceRequestById(Integer) → ServiceRequests** in the Data Control Palette. Remember that this is the collection you used for the top panel on this page (the one that displays the service requests).



3. Drag the **serviceHistoriesCollection** from the Data Control Palette to the last (third) `afh:rowLayout` in the Structure window as an **ADF Read-only Table**.
4. In the Edit Table Columns dialog box, make sure the **Enable selection** check box is **clear**. Click **OK** to accept the defaults.
5. In the Visual Editor, delete all the columns except the **Notes**, **svhType**, and **svhDate** columns. (Click in the column and press [Delete], or right-click and select Delete from the context menu.)
6. Use the Structure window and rearrange the columns into the following order:
   svhDate
   svhType
   notes

   Now you can test your page from the SRList page. You see that the page displays the same row that you chose on the SRList page.
7. Right-click **SRList.jspx** in the Applications Navigator and select **Run**. You can choose to view any of the service requests belonging to you as the signed-on user. Add a note to any service request. Notice that it is displayed in the detail table at the bottom of the page. Close the browser when you are finished testing.

Summary

In this chapter, you created a master-detail page using ADF Faces components. Those components display data that is coordinated between two panels on the page. Without adding any code, the Histories panel shows only those rows associated with the service request displayed in the top panel.

You also added an action that refreshes the page based on a parameter sent from the calling page. In this way, you ensured that the SRMain page displays data that is coordinated across multiple pages.

Here are the key tasks that you performed in this chapter:

- Created the Service Request read-only form
- Created a custom constructor to build a new service histories row
- Created the Notes input form
- Added the Service Request History table

9

Implementing Transactional Capabilities

This chapter describes how to build the pages to create a service request. The service request process involves three main pages: one to specify the product and problem, one to confirm the values, and one to commit and display the service request ID. You also create a fourth page, which displays some frequently asked questions about solving some typical product problems.

The chapter contains the following sections:

- Introduction
- Developing the Create Page
- Creating the Confirmation Page
- Creating the Done Page
- Creating the Frequently Asked Questions Page
- Running the Pages
- Summary

Introduction

Transactional operations for creating and deleting a record are very similar in process. When creating a record, users enter information, click a create button, and then receive a confirmation of the creation. When deleting a record, users select the record, click a delete button, and then receive a confirmation of the deletion. In both cases, an additional step can be included to ask users if they are sure of their action.

In this chapter, you build four screens:

- A page to create or delete a service request record (SRCreate)
- A page to confirm the action, including the newly created service request values (SRCreateConfirm)
- A page to acknowledge the successful creation (SRCreateDone)
- A page to display some static text describing possible service request solutions (SRFaq)

You perform the following key tasks in this chapter:

- Build the Create, Confirm, and Done pages
- Refine the prompts to get their values from UIResources.properties
- Add components for the data-bound objects
- Add command buttons to control transactions
- Manage transactions values
- Pass parameters and navigate between pages
- Define and activate a process train to show the progress of the create process
- Create and link the frequently asked questions

Note: If you did not successfully complete Chapter 8, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter9` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter9`.
2. Unzip `<tutorial_setup>\starterApplications\SRDemo-EndOfChapter8.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter9\SRDemo\SRDemo.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 8.

Developing the Create Page

The SRCreate page enables users to create a new service request. The main menu on the SRList page can call this page. It is also possible to call Create New Service Request on the global menu, which is available on all pages.

SRCreate enables users to select from a list of all appliance products and then enter a description. After entering the description, users can click the Continue button to access a confirmation page (see details on SRCreateConfirm).

However, before users enter the information, they have the option of viewing a Frequently Asked Questions page that, when clicked, presents a modal page that displays a set of frequently asked questions along with accompanying answers (see SRFaq). Another important aspect of the SRCreate page is that users have the option of canceling out of creating a new service request by clicking the Cancel button. Clicking Cancel bypasses all form validation and returns to the SRList page.

Note: Earlier in the tutorial, you created the page outline in the page-flow diagram. In this chapter, you complete the page and apply the template that you created in Chapter 4.

Perform the following steps to build the Create page and attach the template that you created earlier:

1. If it is not open, double-click the `faces-config.xml` file to view the page-flow diagram.
2. Double-click the **SRCreate** page to invoke the JSF Page Wizard.
3. Ensure the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File

Field	Value
File Name	`SRCreate.jspx`
Directory Name	This is the location where the file is stored. The default value should be fine.
Type	`JSP Document`
Mobile	Clear the check box.

4. Click **Next** to continue.

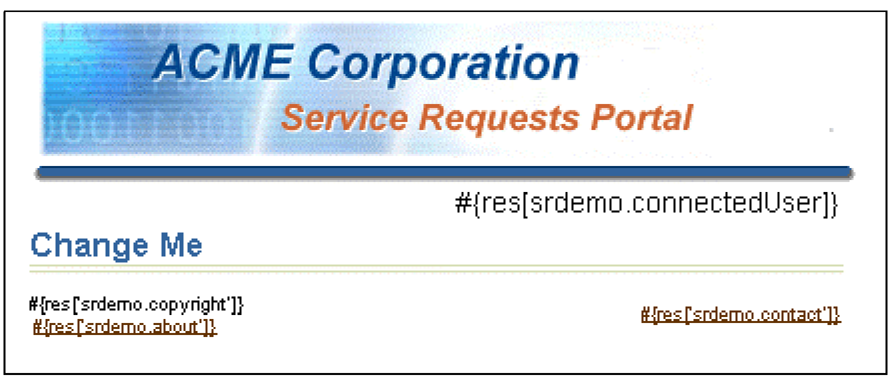
Wizard Step 2: Component Binding	
Field Name	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.
Name	`backing_app_SRCreate`
Class	`SRCreate`
Package	`oracle.srdemo.userinterface.backing.app`

.

5. Click **Next** to continue

Wizard Step 3: Tag Libraries	**Value**
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

6. Click **Finish** to create the page details. The new SRCreate page is displayed in the Visual Editor.
7. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, right-click the **afh:html** node, and choose **Copy** from the shortcut menu.
8. Click the tab to return to the **SRCreate** page, and in the Structure window expand the **f:view** node.
9. Delete the **html** node. Then right-click **f:view** and choose **Paste** from the shortcut menu.
10. In the Structure window, double-click the **afh:head** node and change the Title property to **`SRDemo Create`**.
11. Click **OK**. The Visual Editor now displays the SRCreate page with the look and feel of the other pages.



Refining the Create Page

In this section, you modify the SRCreate page structure to display a different title, change the layout, and include some output items. Because creating a service request is a process, it is useful

to track that process.

1. In the Structure window, expand **the f:view→afh.html→afh.body→h:form** nodes to expose the af:panelPage.
2. Double-click the **af:panelPage** and change the Title property to **`#{res['srcreate.pageTitle']}`**.

   You can set this property to the literal text of your choice for the title. You are setting the property to a value in `the UIResources.properties` file, which contains a list of paired properties and values. At run time, this file is read and the values replaced on the page. The page title will be "Create a New Service Request."
3. Create a Process Train to track the process of creating a service request. From the ADF Faces Core category, drag a **ProcessTrain** onto the af:panelPage component. This enables you to visually see the progress of the create process through all the pages.
4. Select the **af:processTrain**. In the Property Inspector, set the properties to those in the following table:

Field	**Value**
Value	`#{createTrainMenuModel.model}`
Var	`train`
Id	`createStepsTrain`

5. In the Structure window, expand the **af:processTrain → ProcessTrain facets**, exposing the **nodeStamp**. From the ADF Faces Core category, drag a **CommandMenuItem** to the nodeStamp.
6. Select and modify the **af:commandMenuItem** properties to match the table below:

Field	**Value**
Text	`#{train.label}`
Action	`#{train.getOutcome}`
Id	`trainNode`

7. With af:commandMenuItem selected, click the Source tab and set one more property: **`readOnly`**`="#{createTrainMenuModel.model.readOnly}"`.

```
<af:commandMenuItem text="#{train.label]"
                    binding="#{backing_app_SRCreate.trainNode}"
                    id="trainNode"
                    action="#{train.getOutcome}"
                    readOnly="#{createTrainMenuModel.model.readOnly}"/>
```

8. Click the **Design** tab. In the Structure window, drop an **ObjectSpacer** from the ADF Faces Core category. It will be displayed below af:processTrain.

   To do this, drag the **ObjectSpacer** to the af:panelPage, right below the af:processTrain. If you need to move it to a different location, drag it to another node and it will be added at

the bottom of the list in that node. Then clear the **Width** property.

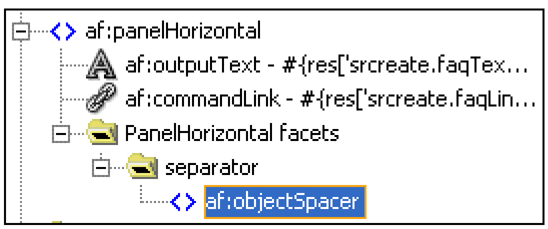
9. To add some extra space to the page and provide a place for some items, drop a **PanelHorizontal** component on the af:panelPage. When you drop the PanelHorizontal on the af:panelPage, it is added to the bottom of the list, below the af:objectSpacer.

10. Expand the **af:panelHorizontal** node. From the ADF Faces Core category, drop an **OutputText** item and a **CommandLink** inside the af:panelHorizontal node. The output text and command link items display information about checking the Frequently Asked Questions page to solve a problem.

11. Set the af:outputText: Value property to **`#{res['srcreate.faqText']}`**. Set the af:commandLink properties to those in the following table:

Field	**Value**
Text	`#{res['srcreate.faqLink']}`
Action	`FAQ`
PartialSubmit	`true`

12. Create some extra space between the text and link. In the af:panelHorizontal node, drop an **ObjectSpacer** on the PanelHorizontal facets →separator node. Set the Width of the spacer to **`4`**. The Structure window should look like the following screenshot:



You now add some explanatory text about what to do on this page and test it.

13. Collapse the **af:panelHorizontal** node and add another **ObjectSpacer** below the af:panelHorizontal. To do this, drop the **ObjectSpacer** on the af:panelPage. Clear the **Width** property.

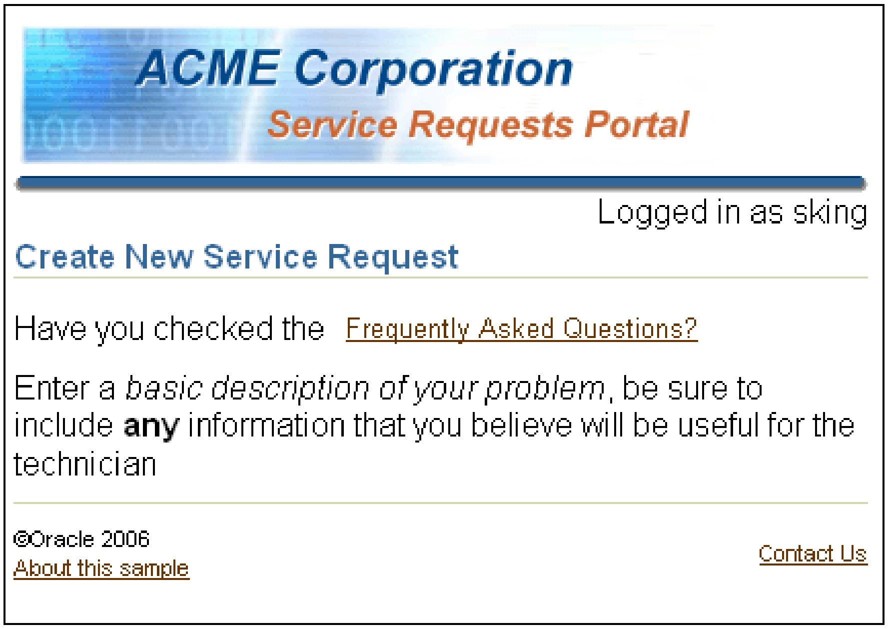
14. Below the af:objectSpacer, add an **OutputFormatted** item from the ADF Faces Core category.

15. Double-click the **af:outputFormated** item and set the Value property to **`#{res['srcreate.explainText']}`**. This displays text on the page to explain how to enter information about the service request.

16. Add another **ObjectSpacer** below the af:outputFormatted item. Clear the **Width** property.

17. In the Visual Editor, select **Run** from the context menu. When the page runs, it displays the structure of the form with headers and labels derived from the `UIResources.properties` file.

18. When prompted, use **`sking`** as the username and **`welcome`** as the password.



Adding Data Components to the Page

In this section, you include the data components for creating a service request. The input form on SRCreate enables users to select from a list of appliances, which is populated from the Products table.. Users are also presented with a text area to enter the problem description. When they click the Continue button, an action of "Confirm" executes and navigates to the SRCreateConfirm page, where the new service request can be committed to the database.

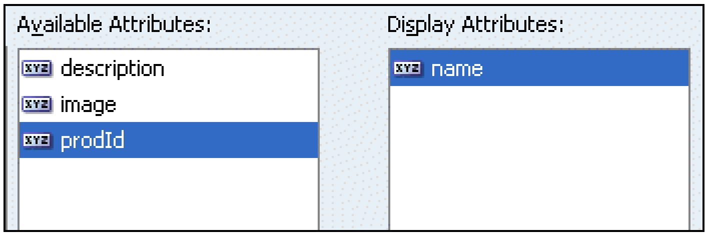
Note: Whenever you want to add a component to the end of a node, drop it on the parent node. JDeveloper automatically adds it to the end of the list. After it is added, you can move it.

---

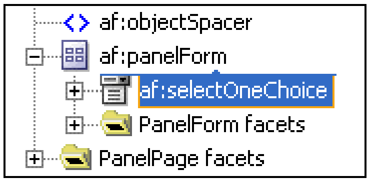
1. The first thing you need is a panel to contain the data components. In the ADF Faces Core category, drag a **PanelForm** to the Structure window, and drop it so it is below the last af:objectSpacer. To do this, drop the PanelForm onto the af:panelPage.
2. Click the **Data Control Palette** tab. Expand the **SRPublicFacadeLocal → findAllProducts()** nodes and select the return value node **Products**.



3. Drag this component to the af:panelForm. In the pop-up menu, select Navigation > ADF Navigation List.
4. In the List Binding Editor, make sure only the name attribute is in the list of Display Attributes. Click OK to continue.



5. The list is created in a af:panelGroup, which we don't need. In the Structure window, drag **af:selectOneChoice** to the af:panelForm. Then select and delete the **af:panelGroup**.



   This action also deletes the nested af:panelHeader labeled Details in the af:panelGroup.

6. You want to show the list of Products as a T-List, not a drop-down list. So select the **af:selectOneChoice** again and from the context menu choose **Convert**. Scroll down and choose **SelectOneListbox** from the dialog box and click **OK**.



7. In the Property Inspector, for the af:selectOneListbox item, set the Label property to **`#{res['srcreate.info.1']}`**, which is obtained from UIResources.properties. Also set the AutoSubmit property to **`false`**. This property controls what happens when you select one value. If it is true, the value you select is immediately submitted. If false, then it is not submitted until you explicitly perform a submit action.

8. From the ADF Faces Core category; drag an **InputText** to the af:panelForm. It will be displayed after the af:selectOneListbox. Set its values to those from the following table. Then click **OK**.

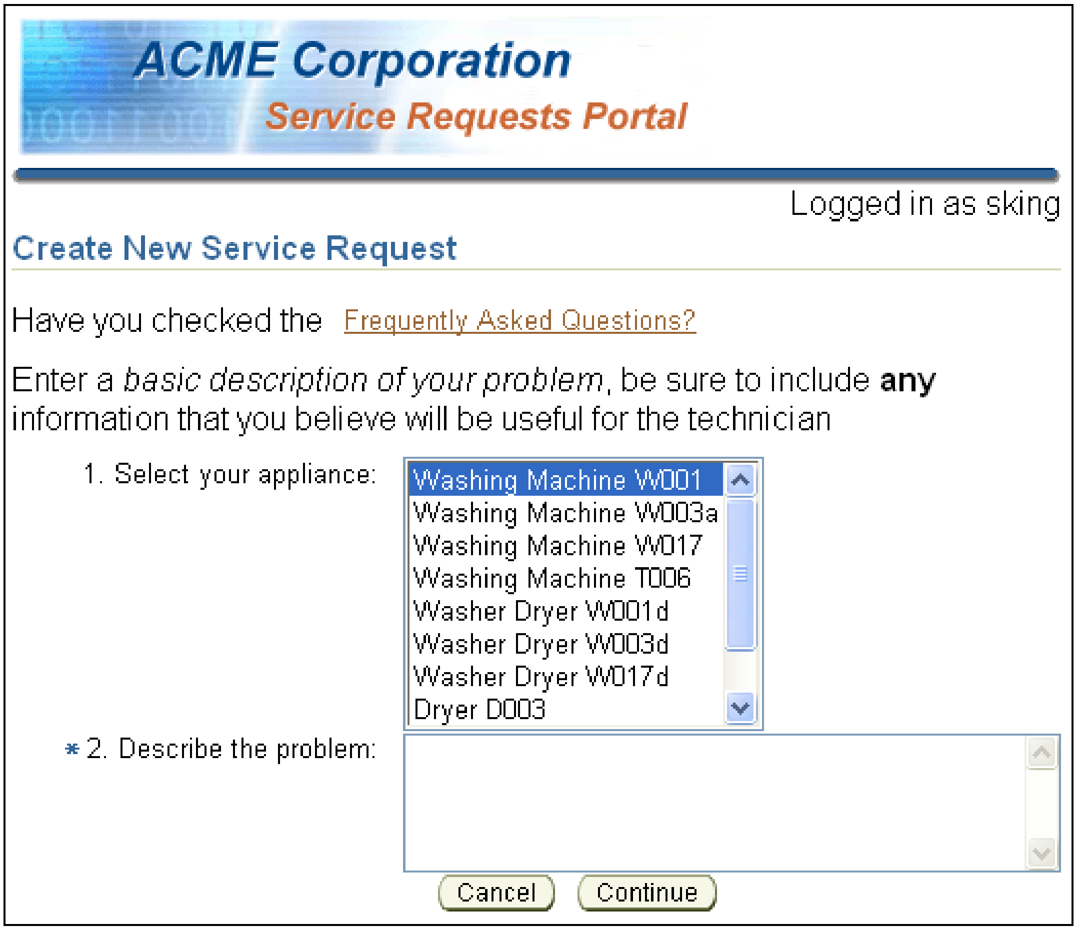
Field	Value
Label	`#{res['srcreate.info.2']}`
Columns	`50`
Rows	`4`
Required	`true`

9. Drop a **PanelButtonBar** onto the af:panelForm→PanelForm facet→footer.

10. Drop a **CommandButton** onto the af:panelButtonBar for users to cancel the creation of a service request. Set the Text property to **`#{res['srdemo.cancel']}`**, set Immediate to **`true`**, and set Id to **`cancelButton`**.

11. Drop a second **CommandButton** onto the af:commandButtonBar for users to continue creating the service request. Set its Text property to **`#{res['srdemo.nextStep']}`** and set the Action property to **`confirm`**.

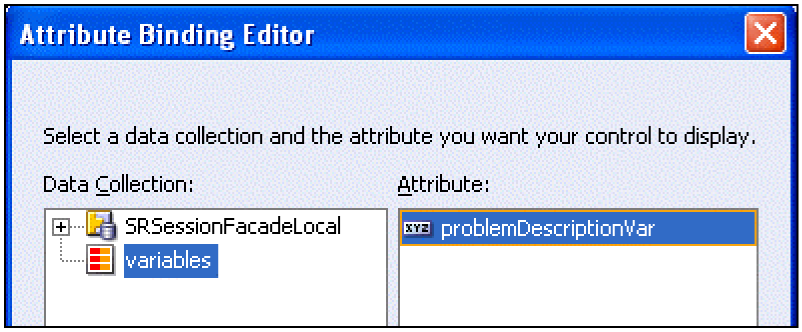
12. Click **Save**, and then click **Run** to run the page.



Saving the Problem Description Value

On this page, the user basically does two things: selects a product from the list and enters the problem description. You "record" the currently selected product by using the current row in the results collection. This happens because you created the item from the Data Control palette. However, the Problem Description field is unbound. You can save its value by using a variable.

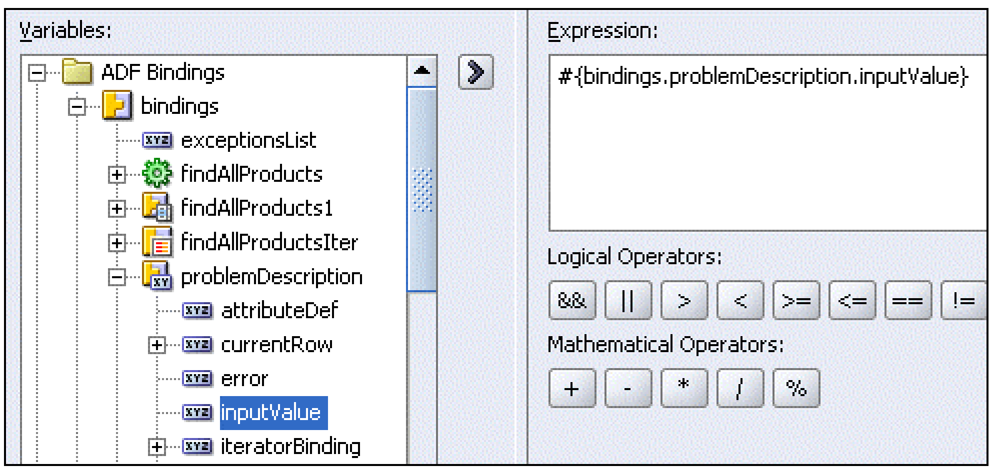
1. Save the variable in the page definition file for the SRCreate page. From the Visual Editor, use the **Go to Page Definition** context menu and navigate to the file.
2. Right-click **executables** node and select **Insert inside executables|variableIterator**.
3. In the Structure window, expand the **executables** node. Select the **variables** node and from the context menu choose **Insert inside variables | variable**. Set the Name of the variable to `problemDescriptionVar` and the Type to `java.lang.String`. Click **OK**.
4. Select **bindings** in the Structure window and **Insert inside Bindings | attributeValues**.
5. In the Attribute Binding Editor, select the **variables** node and the problemDescriptionVar should display on the Attribute panel. Select it and confirm, at the bottom of the page, that the "Select an Iterator" drop-down displays variables. Click **OK**.



6. In the Structure window, select **bindings → problemDescriptionVar**. Change the `ID` property from `problemDescriptionVar` to **`problemDescription`**.

7. **Save** your work.

8. Click the **SRCreate.jspx** tab to go back to the page in the Visual Editor. Select the **af:inputText -#{res['srcreate.info.2']}** field for the problemDescription. In the Property Inspector, use the EL expression picker to set the Value property to #**{bindings.problemDescription.inputValue}**.



Creating Reusable Methods

When you click the cancel button, you want to return to the SRList page (via the GlobalHome navigation rule). When you return to the SRList page, you also want to reset the product list to the first value and clean out the `problemDescription` field. The page is then ready to create another service request.

1. To wire the button, in the Visual Editor double-click the first button in the footer (**#res['srdemo.cancel']**) to define an action. In the **Bind Action Property** pane, accept the default **Method** name (which should be **`cancelButton_action`**), and click **OK**.

2. You are passed into the **SRCreate.java** file. Add the following code to the event. Add an import statement to the beginning of the file:

```
import oracle.binding.BindingContainer;
```

3. This first section of code goes after the variable declarations and before the first method in the class. It defines a variable for the BindingContainer and defines get and set methods for the BindingContainer.

```
private BindingContainer bindings;
```

```
public BindingContainer getBindings() {
    return this.bindings;}
public void setBindings(BindingContainer bindings) {
    this.bindings = bindings;}
```

4. This next piece of code replaces the generated `cancelButton_action()` method. There are inline comments that explain what each section of code does. Replace the existing method with the following:

```
public String cancelButton_action() {
/*
This action is actually reused from two pages, so we just need to ensure that we
use the correct binding container reference.
*/
DCBindingContainer dcBindings =
(DCBindingContainer)ADFUtils.findBindingContainer(getBindings(),
   "app_SRCreatePageDef");
/*
Reset the product list to the first item:
*/
DCIteratorBinding productsIter =
dcBindings.findIteratorBinding("findAllProductsIter");
productsIter.setCurrentRowIndexInRange(0);
/*
Clean out the description field:
*/
AttributeBinding problem =
    (AttributeBinding)dcBindings.getControlBinding
("problemDescription");
problem.setInputValue(null);
/*
Navigate back to the list page
*/
return "GlobalHome";
}
```

5. Include all the import statement in the file by using **<alt><Enter>**. For the AttributeBinding package, import **oracle.adf.model**.
6. **Save** your work.

Assigning a Binding to the Managed Bean

The data components we added to the page were manually bound to their data sources. You now need to create bindings for the SRCreate managed bean. The managed bean enables the UI components to communicate with the data model. ADF created the managed bean for you, so you now need to define a property to manage the binding and then manually bind it.

1. Open the **faces-config.xml** file and click the **Overview** tab.
2. Select the **backing_app_SRCreate** managed bean. In the Managed Properties pane, click the **New** button.
3. Name the property **`bindings`** and click OK.
4. Click the Edit button and set the Value property to **`#{bindings}`**. Click **OK** to complete the process.

Creating the Confirmation Page

The SRCreateConfirm page enables a user to confirm a newly created service request and commit it to the database. It is called when the user clicks the Continue button on the SRCreate page. It displays the new service request information and has three buttons: Cancel, Go Back, and Submit Request. Clicking Cancel cancels the entire new service request entry and navigates to the SRList page. Clicking Go Back returns the user to the SRCreate page but preserves the new tentative service request entry.

> **Note:** In some cases, labels of fields will be pulled from the general resource bundle rather than being inherited from the bindings for that field. All of the labels which contain "#{res[...]}" reference the component attributes.

Here, you continue to develop pages to support the creation process. The second page in the process is the SRCreateConfirm page, which also uses the template you created earlier.

1. With the **`faces-config.xml`** file open, double-click the **SRCreateConfirm** page to invoke the JSF Page Wizard.
2. Ensure that the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File

Field	Value
File Name	`SRCreateConfirm.jspx`
Directory Name	This is the location where the file is stored. Because you last set this value to have an appended **`\app`**, the default value should already contain it. If not, add it.
Type	`JSP Document`
Mobile	Clear the check box.

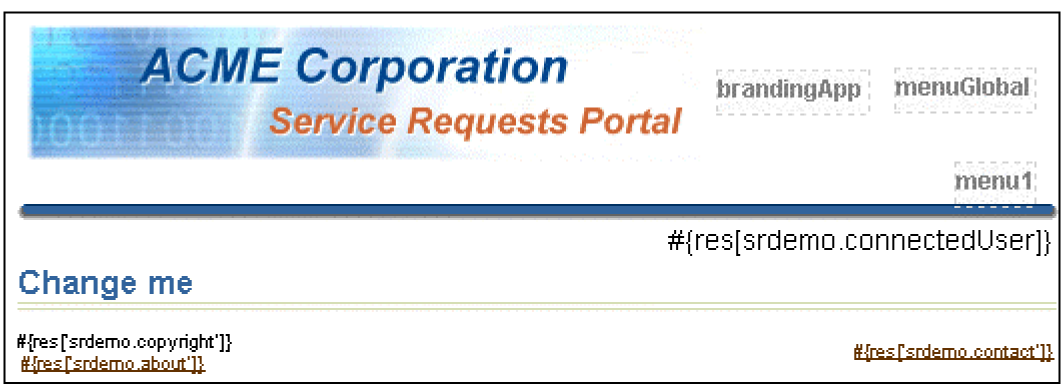
3. Click **Next** to continue.

Wizard Step 2: Component Binding	
Field	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.
Name	`backing_app_SRCreateConfirm`
Class	`SRCreateConfirm`
Package	`oracle.srdemo.userinterface.backing.app`

4. Click **Next** to continue.

Wizard Step 3: Tag Libraries	Value
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

5. Click **Finish** to create the page details. The new **SRCreateConfirm** page is displayed in the Visual Editor.
6. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, right-click the **afh:html** node, and from the shortcut menu, choose **Copy**
7. Click the tab to return to the **SRCreateConfirm** page, and in the Structure window expand the **f:view** node.
8. Delete the **html** node. Then right-click **f:view** and choose **Paste** from the shortcut menu.
9. In the Structure window, double-click the **afh:head** node and change the **Title** property to `SRDemo Confirm`.
10. Click **Finish**. The Visual Editor now displays the **SRCreateConfirm** page with the look and feel of the other pages.



Refining the Confirmation Page

In this section, you modify the SRCreateConfirm page to display a different title and include data-bound controls. The page includes a Process Train to visually show the progress through the create process. On the first page, the first process train's first circle will be filled in and the second circle not filled in. On the SRCreateConfirm page, both circles are filled in. The first few steps focus on enabling the process train functionality.

1. In the Structure window, expand the **f:view→afh.html→afh.body→h:form** nodes to expose the **af:panelPage**.
2. In the Structure window, double-click the **af:panelPage** and set the **Title** property to **`#{res['srcreate.pageTitle']}`**. Just like all the other pages in the tutorial, this value is replaced at run time with a value from UIResources.properties.

Next you need to create a ProcessTrain, just as you did for the SRCreate page. Rather than build the ProcessTrain manually, you can use a copy of the af:processTrain you created for the SRCreate page. The ProcessTrain uses the same values on both pages.

3. Open the **SRCreate** page and expose the **af:panelPage**.
4. Select and right-click the **af:processTrain**. Then select **Copy**.
5. Click the **SRCreateConfirm** page, and in the Structure window expose the **af:panelPage** node.
6. Select the **af:panelPage** and, from the context menu, select **Paste**. The af:processTrain is added as a child of the af:panelPage.
7. Click the **Design** tab and, in the Structure window, drag an **ObjectSpacer** to the af:panelPage. It is added below the **af:processTrain**. Set the height to **`20`** and the width to **`<null>`**.

Refining the Display

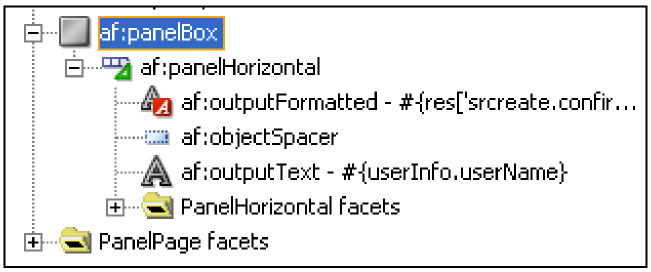
The Process Train is now included to show where you are in the creation process. At this point, you need to display three data values: the user who logged the service request, the product, and the problem description.

In the next steps, you add some confirmation text as well as information about the user creating the service request.

1. Drag an **OutputFormated** component onto the af:panelPage, and it will appear below the af:objectSpacer. Set the Value property to **`#{res['srcreate.confirmText']}`**. At run time, this text asks you if you are confirming the product and problem values.
2. Drop another **ObjectSpacer** and drop it onto the af:panelPage. It should appear below the af:outputFormated component. Set the height to **`20`** and the width to **`<null>`**.
3. Drop a **PanelBox** from the Component palette onto the af:panelPage. It will display after the af:objectSpacer. Set the Width property of the PanelBox to **`100%`**.
4. Drag and drop a **PanelHorizontal** component onto the af:panelBox, making it a child node.

5. Drag an **OutputFormatted** component onto the af:panelHorizontal and set its value to **#{res['srcreate.confirmLine.1']}**.
6. Drag an **ObjectSpacer** component onto the af:panelHorizontal node, and it will display below the af:outputFormated component. Set the width to **<null>** and the height to **20**.
7. Drop an **OutputText** component onto the the af:panelHorizontal . It will display below the af:objectSpacer. This outputText will display information about the user logging the service request. Display the username and the user ID in this component. Set the Value property to **#{userInfo.userName} #{userInfo.userId}**.

Your Structure window should look like the following screenshot. This completes the task of adding information about the user who logged the request.

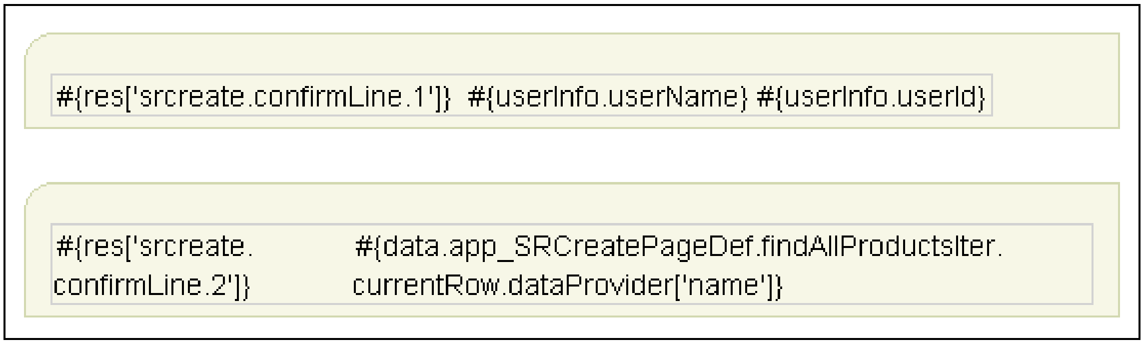


Displaying the ProductId

You now add the product ID to the page. You can copy and paste ADF components from one location to another in the Structure window. The following steps for adding the product ID are similar to those you just completed for the user.

1. Drop an **ObjectSpacer** component onto the af:panelPage, and it will appear below the af:panelBox component. Set the height to **20** and the width to **<null>**.
2. Select the **af:panelBox** you created in the previous step. From the context menu, select **Copy**.
3. Select the **af:panelPage** and, from the context menu, select **Paste**. A new af:panelBox should be displayed below the af:objectsSpacer you just added.
4. Expand **af:panelBox → af:panelHorizontal** and modify the properties of the new components according to the steps below.
5. Set the af:outputFormatted component to **#{res['srcreate.confirmLine.2']}**.
6. Set the af:outputText Value property to **#{data.app_SRCreatePageDef.findAllProductsIter.currentRow.dataProvider['name']}**.

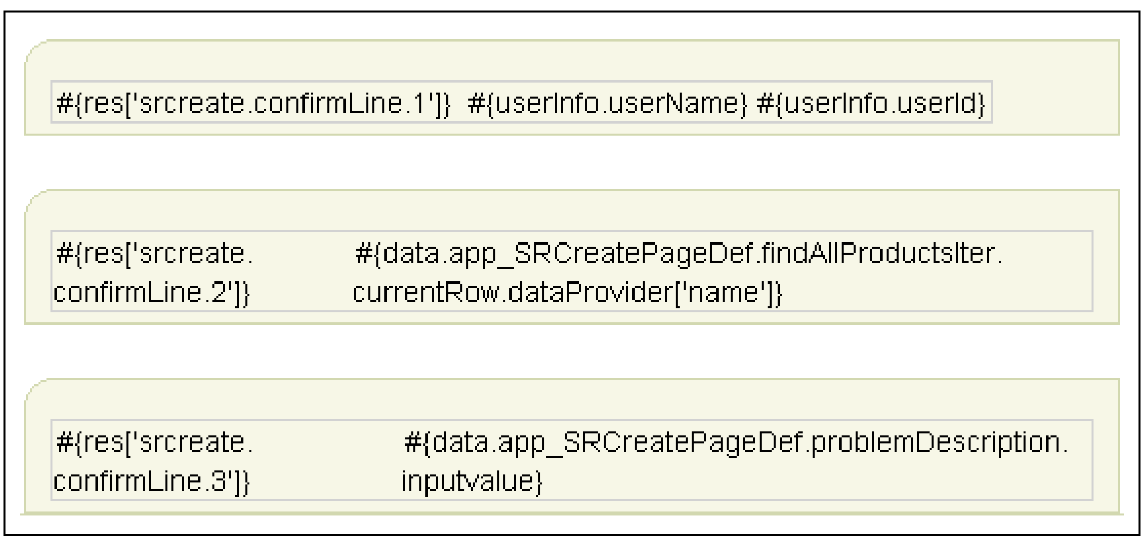
The two areas you just created on the page should look like the following screenshot:



Displaying the Product Description

Finally, you add the product description. You copy and paste ADF components from one of the previous locations and update the properties. The following steps for adding the description are similar to those you just completed for the product ID.

1. Drop an **ObjectSpacer** component onto the af:panelPage. It will appear below the af:panelBox component. Set the height to **20** and the width to **<null>**.
2. Select the **af:panelBox** you created in the previous step. From the context menu, select **Copy**.
3. Select the **af:panelPage** and, from the context menu, select **Paste**. An af:panelBox should be displayed below the af:objectsSpacer you just added.
4. In af:panelHorizontal, set the af:outputFormatted component to **#{res['srcreate.confirmLine.3']}**.
5. Set the af:outputText Value property to **#{data.app_SRCreatePageDef.problemDescription.inputValue}**.
6. Add one final **ObjectSpacer** below the third af:panelBox with height of **20** and width of **<null>**. You have now created the three display items for the page. The Visual Editor should look like the following screenshot:



Controlling the Transaction

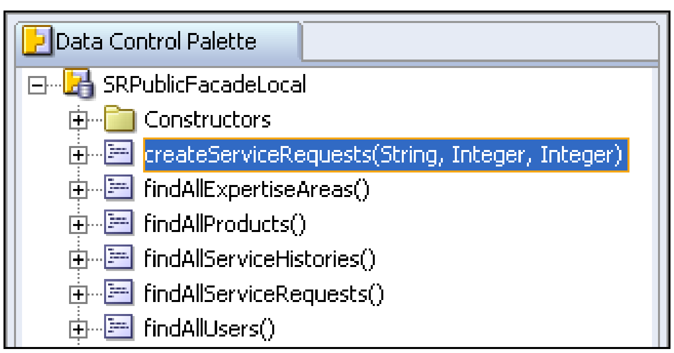
At this point, the page includes components displaying information about the user who logged the service request, the product, and the problem description. You now add some buttons to control the transaction.

1. Click the **SRCreateConfirm.js**px tab to go back to the page.
2. Using the Structure window, drop a **PanelButtonBar** on the af:PanelPage , positioning it after the last af:objectSpacer.
3. Inside the af:panelButtonBar, drop two **CommandButton** components.
4. In the first button, set the following properties: Text to `#{res['srdemo.cancel']}` and Action to `#{backing_app_SRCreate.cancelButton_action}`. (Note how this calls back to the same cancel action as the SRCreate page.)
5. In the second button, set the following properties: Text to `#{res['srdemo.previousStep']}` and Action to `back`. This button navigates you back to the create page.

Binding the Method to the Button

When you click the third button, a few things must happen. The first is to collect the bound values and populate the parameters of the method you want to execute. All the methods from your session bean are available to you from the Data Control method. In the next few steps, you create and bind the third button.

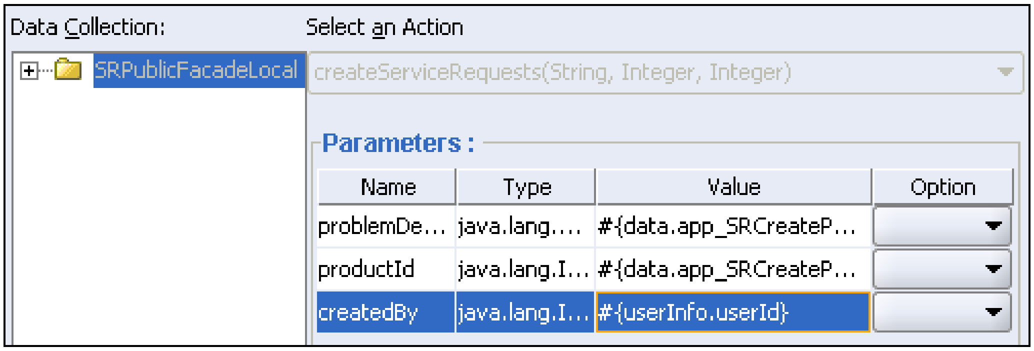
1. Open the Data Control palette. Drag the `createServiceRequests(String,Integer,Integer)` method to af:panelButtonBar. In the pop-up menu, select **Methods|ADF Command Button**.



You assign the parameters to their sources using the Action Editor. Set the binding values for the product ID and description and the user values.

2. The problem description needs to populate the first parameter of the createServiceRequest method. Set the problemDescription Value to `#{data.app_SRCreatePageDef.problemDescription.inputValue}`.
3. The second parameter needs to be the product ID for the product picked from the select one list. Set the productId Value to `#{data.app_SRCreatePageDef.findAllProductsIter.currentRow.dataProvider['prodId']}`. You can add all but the ['prodId'] using the EL picker. You must add the last part manually.

4. The third value is the user ID of the person logged in. Set the createdBy Value to `#{userInfo.userId}`. The Action Editor should look like the following screenshot:

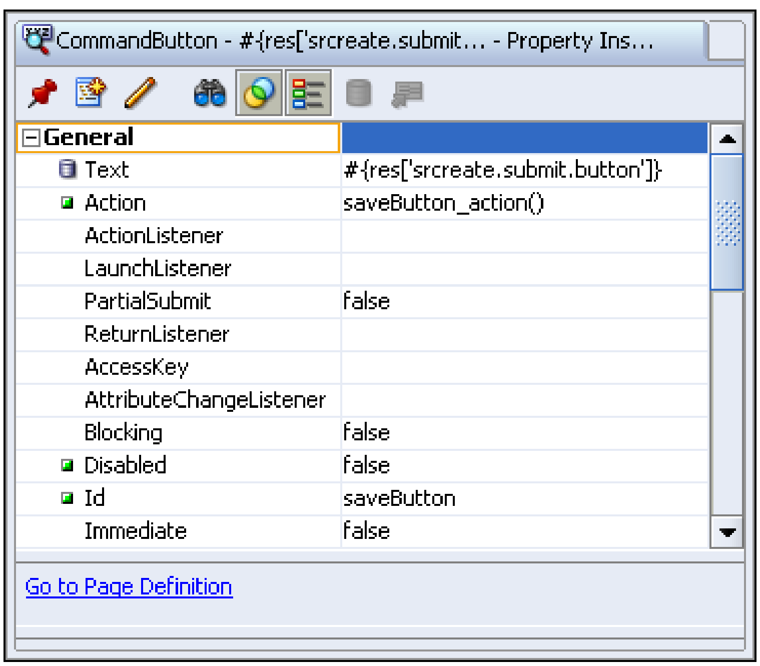


5. Click **OK** to continue.

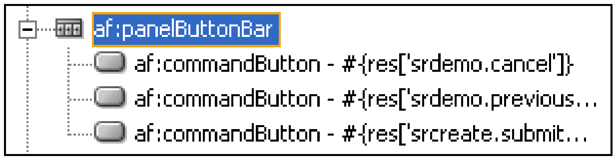
6. Use the following table to set the property values for the third af:commandButton:

Property	Value
Text	`#{res['srcreate.submit.button']}`
Id	`saveButton`
Disabled	`false`
ActionListener	`Should be blank`

The Property Inspector should look like the following screenshot.



7. **Save** your work. The Structure window now includes the three buttons:



Committing the Data and Returning the Service Request ID

The parameters of the `createServiceRequests` method are now bound. Clicking the button must now perform two actions. It must commit the data values to the database. It then must return the service request ID to an attribute for display as confirmation on the SRDone page. The service request ID is generated by the native sequence generator and mapped to the TopLink Service Request definition. In the next series of steps, you add code to the button to accomplish these actions.

1. In the Visual Editor, double-click the third button you created to define an action in the backing bean.



2. In the pop-up Bind Action Property dialog box, accept the default value for the **Method** name. Ensure that the **Generate ADF Binding Code** check box is selected. Selecting this check box adds the import statement and all the getters and setters for the ADF bindings. Click **OK** to continue.
3. You are passed into the **SRCreateConfirm.java** file.
4. To get both actions to happen, replace some of the generated code in the `saveButton_action()` method with the following code. Paste your code below the OperationsBinding line of code.

```
OperationBinding operationBinding =
        bindings.getOperationBinding("createServiceRequests");
//Add this code
ServiceRequests result = (ServiceRequests)operationBinding.execute();
Integer svrId = result.getSvrId();
ExternalContext ectx =
      FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request = (HttpServletRequest)ectx.getRequest();
request.setAttribute("SRDEMO_CREATED_SVRID",svrId);
return "complete";
}
```

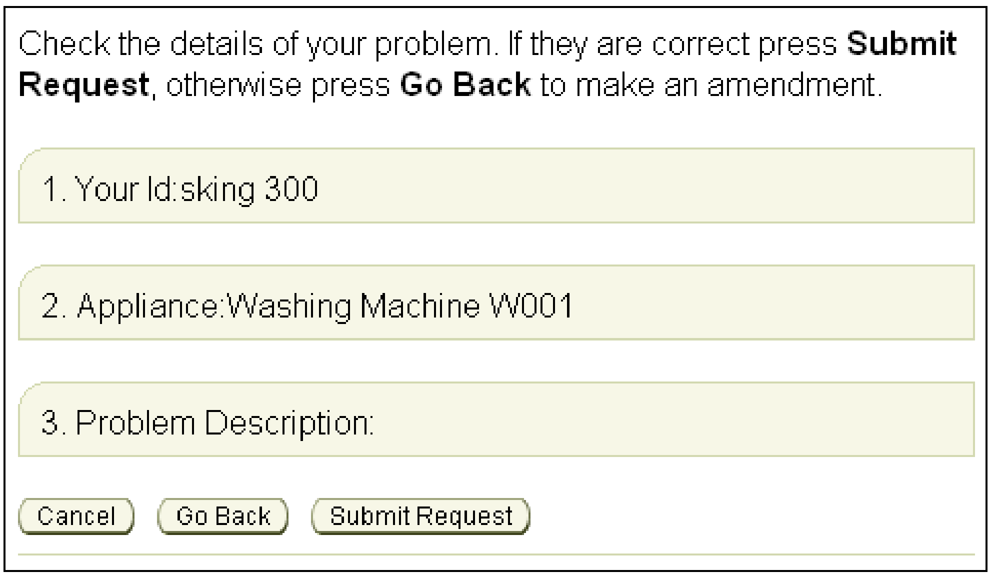
5. Import all the suggested packages by pressing [Alt] + [Enter].

The complete method should look like the following:

```
public String saveButton_action() {
   BindingContainer bindings = getBindings();
   OperationBinding operationBinding =
      bindings.getOperationBinding("createServiceRequests");
   ServiceRequests result =
```

```
            (ServiceRequests)operationBinding.execute();
        Integer svrId = result.getSvrId();
        ExternalContext ectx =
            FacesContext.getCurrentInstance().getExternalContext();
        HttpServletRequest request =
            (HttpServletRequest)ectx.getRequest();
        request.setAttribute("SRDEMO_CREATED_SVRID",svrId);
        return "complete";
    }
```

6. Save your work and Run the page. Because the page runs without being called from the SRCreate page, the product defaults to the first one retrieved from the database, and the problem description is blank. The following screenshot shows the page. When prompted, use `sking` as the username and `welcome` as the password.



Creating the Done Page

The SRDone page informs the user that a service request will be assigned and processed by a technician. It also displays the service request ID for the user's records.

Note: In some cases, a field label is pulled from the general resource bundle rather than inherited from the bindings for that field.

The SRCreateDone page also uses the template that you created earlier in the tutorial.

1. In the `faces-config.xml` file, double-click the **SRCreateDone** page to invoke the JSF Page Wizard.
2. Ensure that the values for the first three steps of the wizard match those in the following tables:

Wizard Step 1: JSP File	
Field	**Value**
File Name	`SRCreateDone.jspx`
Directory Name	This is the location where the file is stored. Because you last set this value to have an appended `\app`, the default value should already contain it. If not, add it.
Type	`JSP Document`
Mobile	Clear the check box.

3. Click **Next** to continue.

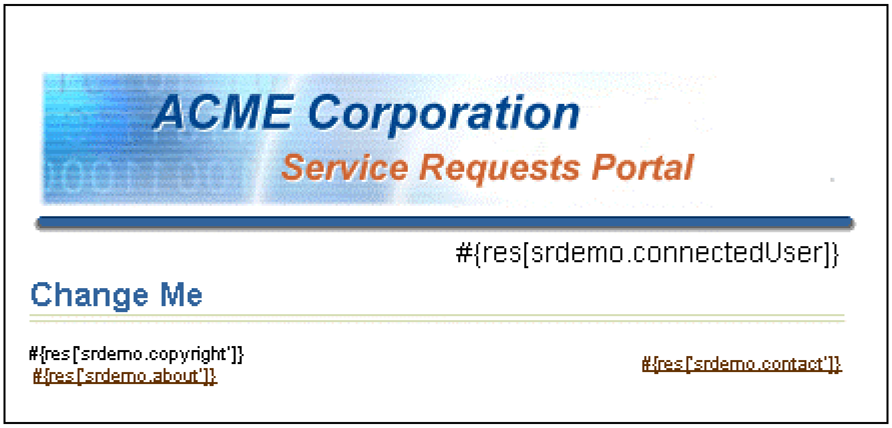
Wizard Step 2: Component Binding	
Field Name	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that this option is selected.
Name	`backing_app_SRCreateDone`
Class	`SRCreateDone`
Package	`oracle.srdemo.userinterface.backing.app`

4. Click **Next** to continue.

Wizard Step 3: Tag Libraries	Value
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

5. Click **Finish** to create the page details. The new SRCreateDone page is displayed in the Visual Editor.
6. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, right-click the **afh:html** node, and from the shortcut menu choose **Copy**
7. Click the tab to return to the SRCreateDone page, and in the Structure window expand the **f:view** node.
8. Delete the **html** node. Then right-click **f:view** and choose **Paste** from the shortcut menu.
9. In the Structure window, double-click the **afh:head** node and change the Title property to **`SRDemo Done`**.

10. Click **OK**. The Visual Editor now displays the SRCreateDone page with the look and feel of the other pages.



Refining the Done Page

In this section, you modify the SRCreateDone page to display a different title, to include a confirmation message, and to display a button that navigates back to the home page. This page is not data bound, although it does use a value from the HTTP request that holds the new SvrId number.

1. In the Structure window, expand the **f:view→ afh.html→ afh.body→ h:form** nodes to expose the **af:panelPage**.
2. Double-click the **af:panelPage** and change the **Text** property to `#{res['srcreate.pageTitle']}`. Click **OK**. The Visual Editor displays the new panel title.
3. In the visual view, drag an **ObjectSpacer** to the af:panelPage. Set the width to `<null>`.
4. Drag a **PanelBox** from the Component palette to the af:panelPage and set the width to `100%`.
5. Drag an **ObjectSpacer** to the af:panelBox.
6. From the Component palette, drag an **JSF HTML → OutputFormat** component to the af:panelBox.

In the next few step, you display a message about assigning the service request to a technician and defining where to go when finished.

7. Select the **h:outputFormat** component and click the **Source** tab. Replace the existing code with the following code. This code includes a JSF parameter for the user who created the service request.

```
<h:outputFormat value="#{res['srcreate.confirmPanel.message']}"
escape="false">

<f:param value="#{requestScope.SRDEMO_CREATED_SVRID}"/>

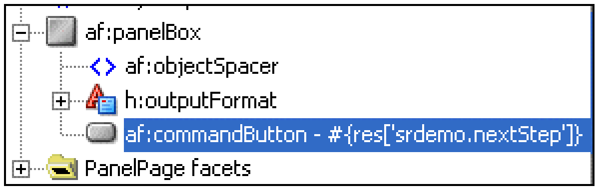
</h:outputFormat>
```

8. Click the **Design** tab and drag an **ADF Faces Core → CommandButton** to the af:panelBox. It will be displayed below the **h:outputFormat** component.

9. Select the button and set the following two properties:

   Text to **`#{res['srdemo.nextStep']}`**
   Action to **`globalHome`**

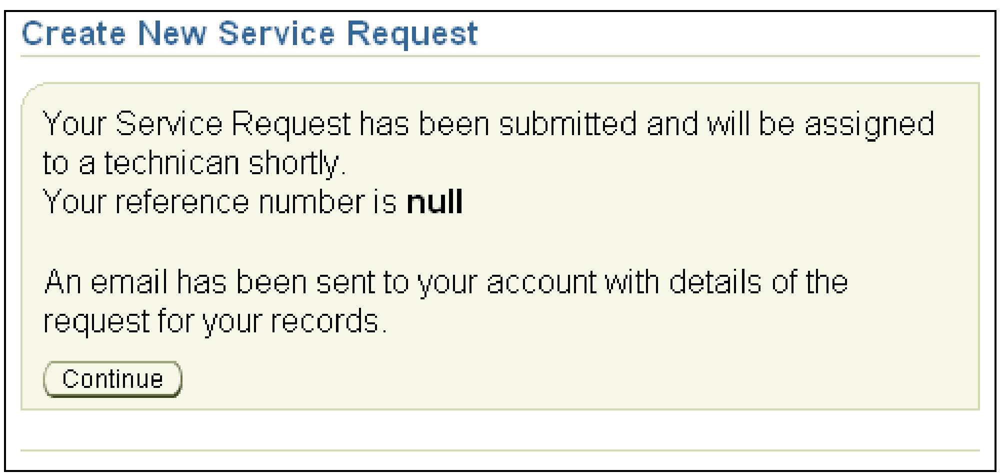
   Your button should look like the following screenshot:



10. To give a bit more space, drag an **ObjectSpacer** to the af:panelPage. It will appear after the af:panelBox. In the Visual Editor, your af:panelBox should look like the following screenshot:



11. **Save** your work and **Run** the page. Because the page is not being run after the first two pages, there is no service request ID. This is why the text referring to the reference number displays "null" when you run the page. If you run the page as the third in the sequence, a value service request ID is displayed. The page should look like the following screenshot. When prompted, use **`sking`** as the username and **`welcome`** as the password.



Creating the Frequently Asked Questions Page

In this section, you develop the SRFaq page, which can be invoked from the SRCreate page. This page displays a few frequently asked questions that might help users solve their problems. The text values are retrieved from the `UIResources.properties` file.

1. In the `faces-config.xml` file, double-click the **SRFaq** page to invoke the JSF Page Wizard. The values should be correct for the first step in the wizard. Click **Next**.
2. In the second step of the wizard, select the **Do Not Automatically Expose UI Components in a Managed Bean** radio button. The only function this page performs is to display static text, so no backing bean is required.
3. Click **Finish**. This creates the page. You need only to add the template and text references.
4. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, right-click the **afh:html** node, and from the shortcut menu choose **Copy**.
5. Click the tab to return to the SRFaq page, and in the Structure window expand the **f:view** node.
6. Delete the **html** node. Then right-click **f:view** and choose **Paste** from the shortcut menu.
7. In the Structure window, double-click the **afh:head** node and change the Title property to **`SRDemo FAQ`**.
8. Click **OK**. The Visual Editor now displays the SRFaq page with the look and feel of the other pages.

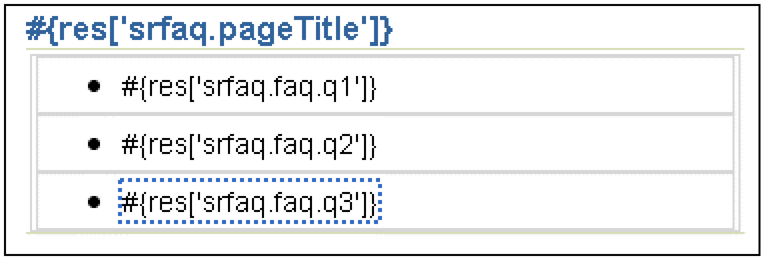
Refining the FAQ Page

In this section, you modify the SRFaq page to display three common service request problem descriptions and a solution for each. The text is retrieved from the `UIResources.properties` file and can be modified there. For the purposes of this page, the text displays three service request problems and their solutions.

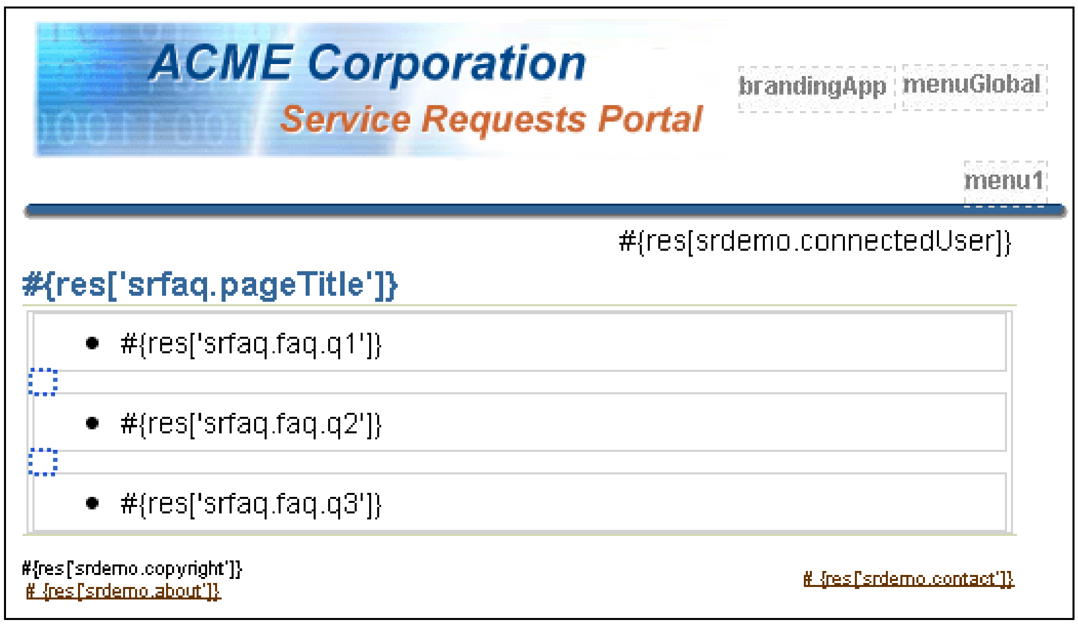
1. In the Structure window, expand the **f:view→ afh.html→ afh.body→ h:form** nodes to expose the **af:panelPage**.
2. Double-click the **af:panelPage** and change the **Text** property to **`#{res['srfaq.pageTitle']}`**. Click **OK**. The Visual Editor displays the new panel title.
3. In the visual view, drag an **ADF Faces Core → PanelGroup** inside the **af:panelPage**.
4. Within the **af:panelGroup**, add three **ADF Faces Core → PanelLists**. These panel lists are used as placeholders for the text you want to display.
5. In each of the **af:panelList** components, add an **ADF Faces Core → OutputFormatted** component. Set the **Value** property in each to the values from the following table. Each of these fields corresponds to a service request solution value found in the `UIResources.properties` file.

Output Formatted item	Value
1st	`#{res['srfaq.faq.q1']}`
2nd	`#{res['srfaq.faq.q2']}`
3rd	`#{res['srfaq.faq.q3']}`

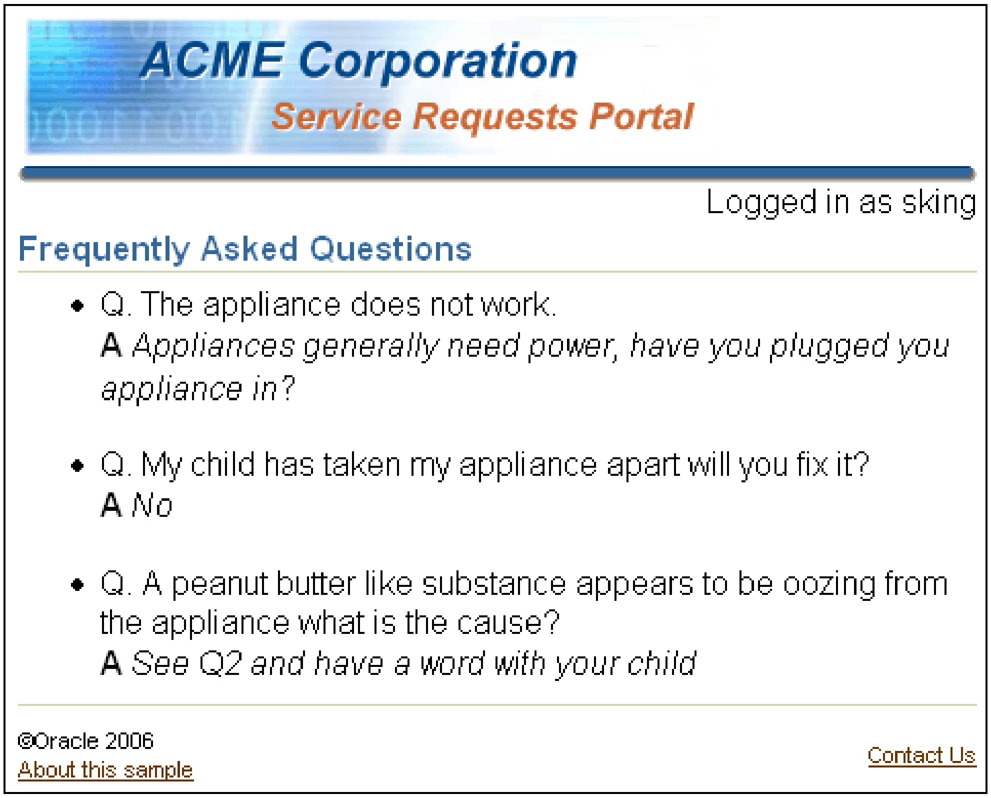
The page components you just added should make the page look like the following screenshot:



6. Expand the **af:panelGroup → PanelGroup facets → Seperator** nodes and add an **ObjectSeperator** within the separator node. This action adds a separator between each of the af:panelist components to give the panel a little extra space. The finished page should look like the following screenshot:



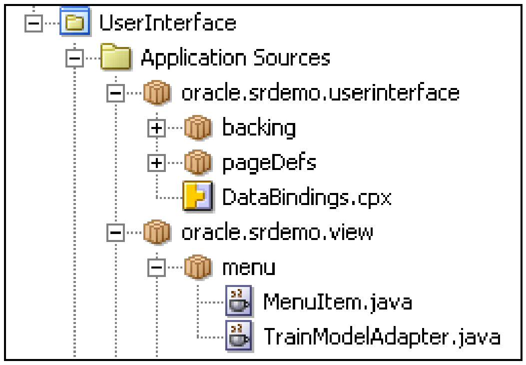
7. **Save** your work and **Run** the page. Your page should look like the following screenshot. When prompted, use `sking` as the username and `welcome` as the password.



Wiring the Process Train

On both the SRCreate and SRCreateConfirm pages, you've added a process train to view the progress though the create process. While developing the pages, you created the display for the process trains. In this section, you create two classes and then create four managed beans and link them to the process train.

1. In the Applications Navigator, expand the **UserInterface → Application Sources** nodes.
2. Select the **oracle.srdemo.view** node and from the context menu, select **New**.
3. In the New Gallery, select **General** and then **Java Class**. Click **OK**.
4. Name the class **`MenuItem`** and set the Package to **`oracle.srdemo.view.menu`**. Then click **OK**.
5. In Windows Explorer, navigate to the location where you unzipped the setup file. Open the **`MenuItem.txt`** file and copy all of it into the new class you just created.
6. Create a second Java class, with the name **`TrainModelAdapter`** and with the Package value set to **`oracle.srdemo.view.menu`**.
7. In Windows Explorer, navigate to the location where you unzipped the setup file. Open the **`TrainModelAdapter.txt`** file and copy all of it into the new class you just created.
8. The Applications Navigator should look like the following screenshot:



9. In the Applications Navigator, double click the **`faces-config.xml`** file and click the **Overview** tab.
10. Create four managed beans for the Process Train using the values in the following table. In each case, set the Scope property to **`application`** and clear the "**Create class if it doesn't exist"** check box.

Name	Class
createTrain_Step1	`oracle.srdemo.view.menu.MenuItem`
createTrain_Step2	`oracle.srdemo.view.menu.MenuItem`
createTrainNodes	`java.util.ArrayList`
createTrainMenuModel	`oracle.srdemo.view.menu.TrainModelAdapter`

Each bean performs a different function. The two createTrain_Step beans connect to a simple bean that represents how the item is displayed. The createTrainMenuModel connects to the class responsible for holding the nodes of the process train.

Each managed bean contains information about the current process step. This includes the page name and the current process step. In the next few steps, you create and assign values to the managed properties.

11. Create managed properties for three of the managed beans. Use the following tables and the faces-config.xml - Overview page to complete this step. For all the Managed Properties, leave the Class property set to **`<null>`**.

    After you create the managed properties listed in the tables, double-click each one and set their values to the values in the second column of the tables.
12. Select the **createTrain_Step1** managed bean and create the following Managed Properties.

Managed Properties	Value
`label`	`#{resources['srcreate.train.step1']}`
`viewId`	`/app/SRCreate.jspx`
`outcome`	`GlobalCreate`

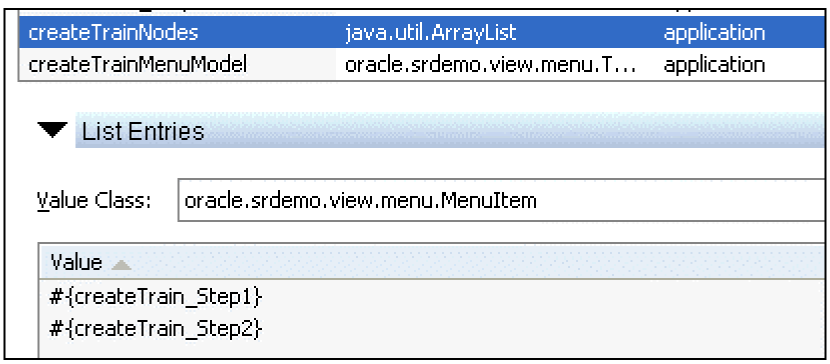
13. Select the **createTrain_Step2** managed bean and create the following Managed Properties.

Managed Properties	Value
`label`	`#{resources['srcreate.train.step2']}`
`viewId`	`/app/SRCreateConfirm.jspx`
`outcome`	`Confirm`

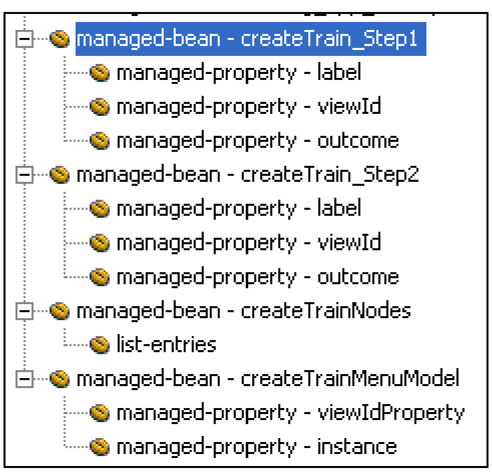
14. Select the **CreateTrainMenuModel** managed bean and create the following Managed Properties.

Managed Properties	Value
`viewIdProperty`	`viewId`
`instance`	`#{createTrainNodes}`

15. Select the remaining **createTrainNodes** managed bean. In the List Entries area, set the Value Class to **`oracle.srdemo.view.menu.MenuItem`** and add two new values: **`#{createTrain_Step1}`** and **`#{createTrain_Step2}`**. When complete, the area should look like the following screenshot:



When you're finished creating all four managed beans, the Structure window should look like the following screenshot:



Running the Pages

Now that the Create page is completed, you can run it and examine the functionality. Here are a

few things to note:

- The SRCreate page enables you to select a product and define a problem.
- The process train is highlighted depending on where you are in the process.
- The product name and problem description values are carried over to the SRCreateConfirm page.
- Your user ID and name are gathered from your login and displayed.
- The Submit Request button navigates you to the SRDone page and displays the new service request ID.
- The Continue button navigates you to the global home, which in your case is the SRList page.
- The new service request record is displayed in the list.

Summary

In this chapter, you built the Search page using JavaServer Faces and ADF components. To accomplish this, you performed the following key tasks:

- Developed the Create page to specify a service request
- Created a Confirm page to display the service request values
- Created a Done page to display the new service request
- Created a Frequently Asked Questions page to display service request solutions

10

Developing an Edit Page

This chapter describes how to create the SREdit page, the page in the SRDemo application that enables managers and technicians to edit service requests.

The chapter contains the following sections:

- Introduction
- Creating the Page Outline
- Adding UI Elements to the Page
- Creating Lookups to Retrieve the `createdBy` and `assignedTo` Names
- Wiring Up the Input Service Request ID Parameter
- Adding a Drop-down List for the `status` Attribute
- Wiring Up the Cancel Button
- Wiring Up the Save Button
- Disabling Input Fields for Closed Service Requests
- Running the Page
- Changing the Application Look and Feel
- Summary

Introduction

The SREdit page is one of three screens intended to be used by the company's staff rather than by its customers. Managers and technicians use the page to update information on service requests.

Note: If you did not successfully complete Chapter 9, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter10` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter10`.
2. Unzip `<tutorial_setup>\starterApplications\SRDemo-EndOfChapter9.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter10\SRDemo\SRDemo.jws`. This opens the starter application for this chapter.

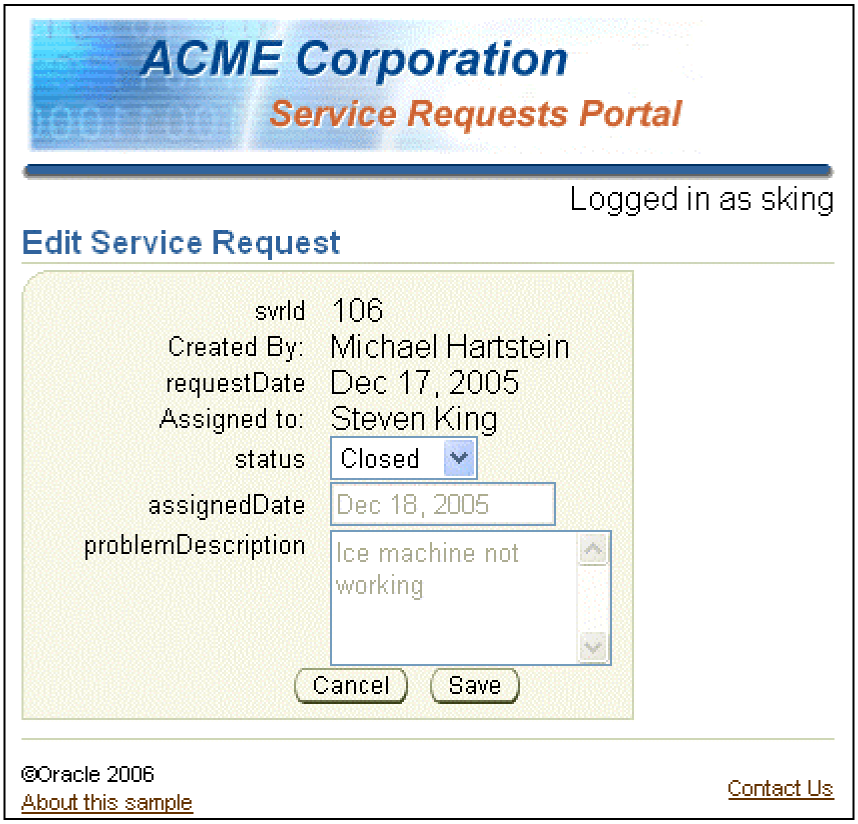
You can now continue with this tutorial using an application that implements all of the steps from Chapter 9.

Double-click the `faces-config.xml` file in the Applications Navigator to revisit the page-flow diagram and examine how the SREdit page relates to the other pages.

Here are some key points to note about the SREdit page:

- The page can be called from three different places within the application: the SRList page, the SRSearch page, and SRMain. In order to reuse the page from these different contexts, it is parameterized; that is, the calling page is expected to set parameters that inform the SREdit page about which service request to query and where to return after the edit is finished.
- The page enables a service request to be edited by a manager or a technician.
- It enables the user to modify the status of a request.
- It provides for the request to be assigned to another user or manager.
- The Problem Description field and the Assigned Date field are disabled for requests with a status of Closed.

The following screenshot shows you how the finished SREdit page should look (from a manager login):



To create the SREdit page with the functionality and look-and-feel described in the preceding list, you perform the following key tasks:

- Create the page outline (based on the template page you created in Chapter 4)
- Add UI elements to the page
- Create lookups to retrieve the `createdBy` and `assignedTo` names
- Wire up the input service request ID parameter
- Add a drop-down list for the `status` attribute
- Wire up the Cancel button
- Wire up the Save button
- Disable input fields for closed service requests

Creating the Page Outline

Perform the following steps to create the SREdit page and add the template to apply the appropriate look and feel.

1. If it is not already open, double-click the **`faces-config.xml`** file to view the **Page Flow Diagram**.
2. Double-click the **SREdit** page to invoke the JSF Page Wizard.
3. Complete the first three steps of the wizard using the values in the following tables:

Wizard Step 1: JSP File	
Field	**Value**
File Name	`SREdit.jspx`
Directory Name	This is the location where the file is stored. Ensure that you create the page in the `\SRDemo\UserInterface\public_html\app\staff` folder.
Type	`JSP Document`
Mobile	Clear the check box.

4. Click **Next**.

Wizard Step 2: Component Binding	
Field	**Value**
Automatically Expose UI Components in a New Managed Bean	Ensure that this radio button is selected.
Name	`backing_app_staff_SREdit`
Class	`SREdit`
Package	`oracle.srdemo.view.backing.app.staff`

5. Click **Next**.

Wizard Step 3: Tag Libraries	
Field	**Value**
Selected Libraries	`ADF Faces Components` `ADF Faces HTML` `JSF Core` `JSF HTML`

6. Click **Finish** to create the page details. The new SREdit page is displayed in the Visual Editor.
7. Open the **SRDemoTemplate** file if it is not already open. In the Structure window, shrink the **afh:html** node and select it. From the context menu, choose **Copy**.
8. Click the tab to return to the SREdit page. In the Structure window, expand the **`f:view`** node.
9. Delete the **`html`** node. Then right-click **`f:view`** and choose **Paste** from the context menu.

10. The main image doesn't appear; you need to change the path. Select the image and then, in the Property Inspector, reset the URL property to **`/images/SRBranding`**.

    The look and feel that you created earlier is now applied to the new page.

Adding UI Elements to the Page

Perform the following steps to start adding some user interface elements to the page. You need to include all the data components from the `findServiceRequestById` method and then add some command buttons to control the save and cancel actions.

1. Add a title to your page as follows: Click the panelPage in the Visual Editor to select it. (Alternatively, you can select **`af:panelPage`** in the Structure window.) In the Property Inspector, type **`#{res['sredit.pageTitle']}`** in the Title property.

   Alternatively, as you've seen several times before in earlier chapters of this tutorial, you can invoke the PanelPage Properties dialog and click **Bind** in the Title property and choose **sredit.pageTitle** from the res node, and shuttle it into the Expression pane

   The name of the page title comes from a resource as defined in the `UIResources.properties` file.

2. Add a header that will appear in the browser title when you run the page, as follows: In the Structure window, select **`afh:head`**. Set the Title property to **`#{res['srdemo.browserTitle']}`**.

3. Select the **ADF Faces Core** page in the Component palette. Scroll through the list to find **PanelBox** and drag it into the panelPage. In the Visual Editor it should display under the title. The panelBox will define the content area for the page.

In the next few steps, you create the data components, resize the field display, and include two command buttons.

4. Now add the data to the page. Select the Data Control palette and expand **SRPublicFacadeLocal**. Scroll down to select **findServiceRequestById(Integer)→ServiceRequests** and drag it to the page, inside the panelBox.

5. In the pop-up menu, select **Forms → ADF Form**. In the Edit Form Fields dialog box that appears, reorder the columns as follows: **svrId**, **requestDate**, **status**, **assignedDate**, and **problemDescription**. Do not close the dialog box yet.

6. Still in the dialog box, click the **Component to Use** field to the right of `svrId`, and from the drop-down list, select **ADF Output Text w/Label**. Do the same for the **`requestDate`** field. Click **OK**.

7. The Action Binding Editor appears. In the dialog box, leave the value property for **findSvrId** empty for now; you will assign a page parameter to it later. Click **OK**.

   The form is displayed on the page. In the next steps, you add details to it:

8. Select the **problemDescription** field and, in the Property Inspector, set the **Rows** property to **4** to make this a multiline item so that a user has space to provide a textual description of the appliance problem.

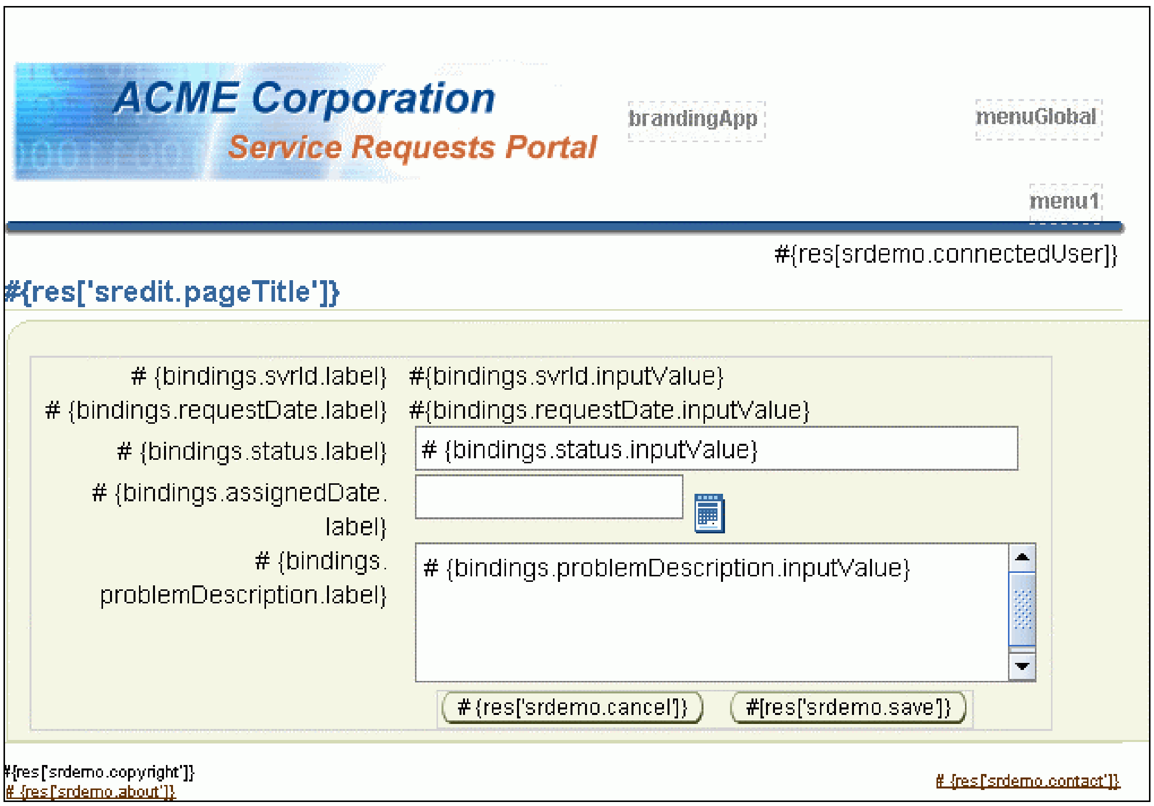
9. Expose the PanelForm facets node. Select **panelButtonBar** in the ADF Faces Core Component palette, and drag it to the footer facet of the PanelForm in the Structure window

   You now create Save and Cancel buttons for the page:

10. Drag two **CommandButtons** from the ADF Faces Core Component palette to the panelButtonBar. In the Property Inspector, set their properties to the values in the following table:

ID	Text
cancelButton	`#{res['srdemo.cancel']}`
commitButton	`#{res['srdemo.save']}`

At this point, your page should look something like the following screenshot:



Creating Lookups to Retrieve the `createdBy` and `assignedTo` Names

The `createdBy` and `assignedTo` attributes enable a user to retrieve the name of the individual who created a service request or the individual to whom a service request is assigned.

The `createdBy` and `assignedTo` attributes of the ServiceRequest object are actually child-object accessors. In the following steps, you bind the relevant name attributes from these child objects into the page.

1. The first component contains the first and last names of the user who created the service request. Select **PanelLabelAndMessage** in the ADF Faces Core Component palette, and in the Visual Editor, drag it to the page beneath the `svrId` field.

2. The second component contains the first and last names of the technician to whom the service request is assigned. Drag and drop a second **PanelLabelAndMessage** component beneath the `requestDate` field.

3. Drop a **PanelHorizontal** inside each af:panelLabelAndMessage component. This ensures that the `firstName` and `lastName` attributes display side by side instead of vertically.

   In the next few steps, you add and bind data components for the first and last name of the created by and assigned to users to the user interface:

4. In the Data Control palette, expand the **findServiceRequestById→ServiceRequests→createdBy** node, and select the **`firstName`** attribute. Drag it to the first of the panelHorizontals, and drop it as ADF Output Text.

5. Repeat the previous step with the **`lastName`** attribute, adding it to the same af:panelHorizontal.

6. Select an **ObjectSpacer** in the ADF Faces Core Component palette. Drag and drop it onto the PanelHorizontal separator facet of the Structure window (expand the PanelHorizontal facets node if necessary). Set its Width property to **`4`**.

7. In the Data Control palette, expand the **findServiceRequestById→ServiceRequests→assignedTo** node, and select the **firstName** attribute. Drag it into the second of the panelHorizontals, and drop it as ADF Output Text.

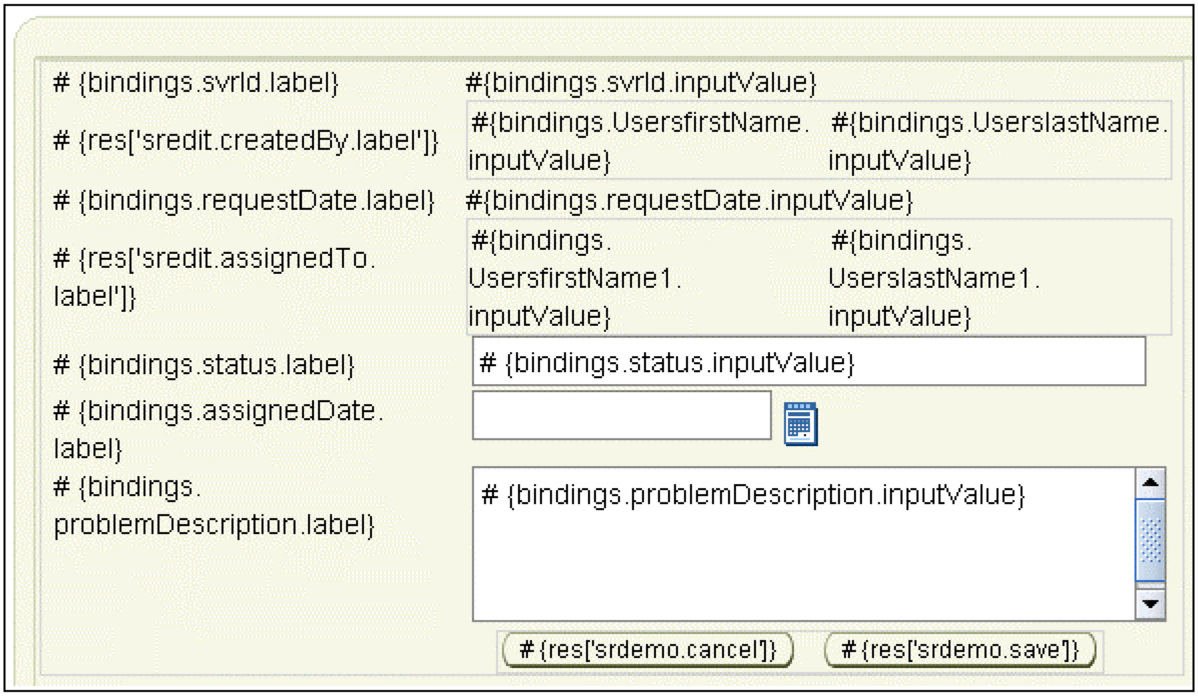
8. Repeat the previous step with the **lastName** attribute, placing it in the af:panelHorizontal component..

9. Add an **ObjectSpacer** to the separator facet of this PanelHorizontal as you did earlier. Again, set its width to **4**.

10. Select the first **panelLabelAndMessage** component. Set the Label property to **`#{res['sredit.createdBy.label']}`**.

11. Set the Label property of the second panelLabelAndMessage component to **`#{res['sredit.assignedTo.label']}`**.

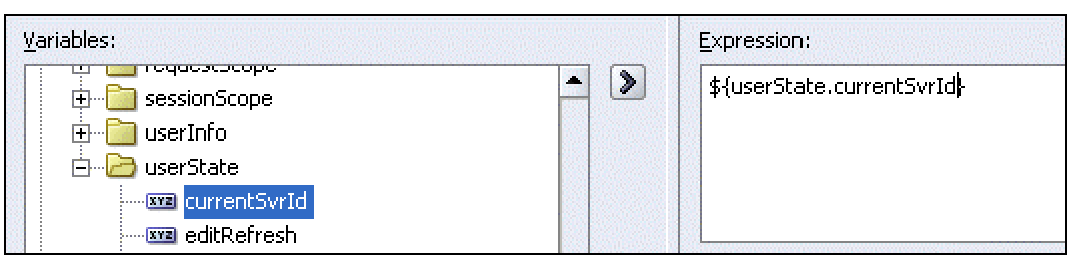
12. Save the page. At this point, the page should look something like the following screenshot:



Wiring Up the Input Service Request ID Parameter

In the following steps, identify and wire the service request that the SREdit page should display when a user enters the page.

1. Right-click in the Visual Editor and, from the context menu, choose **Go to Page Definition**.
2. In the Structure window, expand the **bindings** node, and then expand the **findServiceRequestById** node. Double-click **findSvrId**, which represents the argument for the method.
3. In the NamedData Properties dialog box, click **[...]** to go to the Advanced Editor. In the Variables list, expand the **JSF Managed Beans** node and then the **userState** node, and scroll down to find **currentSvrId**.
4. Select **currentSvrId** and click the **>** arrow to shuttle it to the Expression box. Click **OK**, and then click **OK** again. Save the page definition file.

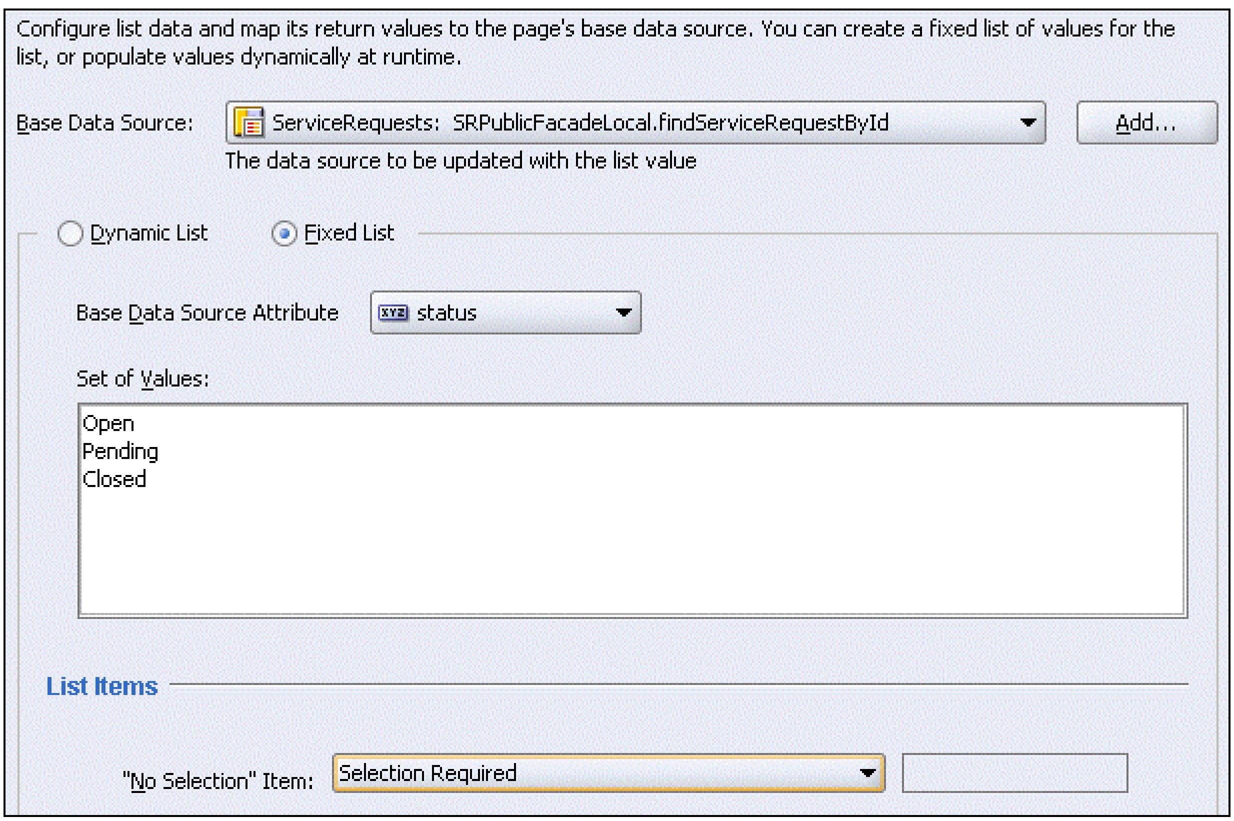


Adding a Drop-down List for the `status` Attribute

The status field should offer the user a drop-down list of the different statuses available. Perform the following steps to convert the current status field to a list:

1. Click the **SREdit.jspx** tab and, in the Visual Editor, delete from the page both the label and input text for the existing status field.
2. In the Data Control palette, expand **findServiceRequestById→ServiceRequests**, and select the **status** attribute. Drag it to its old position in the page, and drop it as Single Selections→ADF Select One Choice.

   The selectOneChoice component creates a menu-style component that enables the user to select a single value from a list of items.
3. In the List Binding Editor, select the **Fixed List** option. Set the Base Data Source Attribute to **status**.
4. In the Set of Values box, enter the following values, each on a new line: **`Open`**, **`Pending`**, and **`Closed`**. These values are displayed at run time.
5. Set the "No Selection" Item field to **Selection Required**. Click **OK**.



6. In the Property Inspector, for the new af:selectOneChoice, set the ID property of the new list component to **`statusSelection`**.
7. Save the page.

Wiring Up the Cancel Button

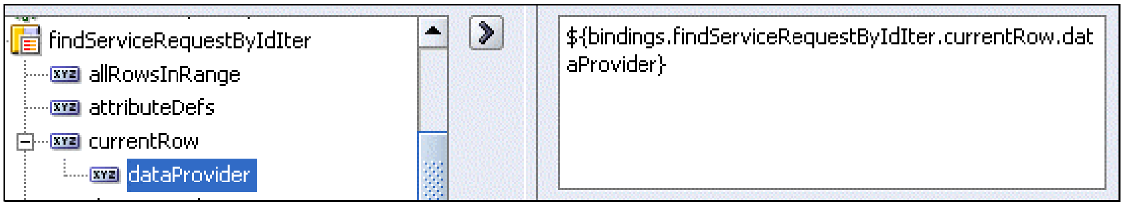
When a user clicks the Cancel button on the Edit screen, you discard the changes by not "saving" them. You then need to take the user back to wherever they came from. You do this by passing a parameter from the calling page. In addition, you need to ensure that if the user returns to the page to view the same service request, the query is reexecuted. Otherwise, the previous state of edit will be displayed.

1. Right-click in the Visual Editor, and choose **Go to Page Definition** from the context menu.
2. In the Structure window, right-click the **executables** node, and choose **Insert inside executables-->invokeAction**
3. In the **Insert invokeAction** dialog box, type `explicitRefresh` as the **id**, and in the **Binds** field choose **findServiceRequestById** from the drop-down list. Do not exit the dialog box.
4. Click the **Advanced Properties** tab, and confirm that the Refresh property is set to **IfNeeded**. This controls whether a refresh should take place.
5. In the RefreshCondition property, click **[…]**. Then expand the **JSF Managed Beans** node and the **userState** node in the Variables pane.
6. Locate and select **refresh**, and then click **>** to shuttle it to the **Expression** pane. Click **OK** to close the editor, and then click **OK** again to exit the dialog box.
7. Save the page definition file.
8. Set an actionListener to fire when the user clicks the Cancel button, and trigger the refresh.
   On the SREdit page, right-click the **Cancel** button and, from the context menu, choose **Insert inside af:commandButton - #{res['srdemo.cancel']}→ADF Faces Core→SetActionListener**.
9. In the Insert SetActionListener dialog box, type `#{true}`in the From* field and `#{userState.refresh}` in the To* field. When fired, this listener populates the user state refresh property with the value of #{true}. Click **OK**.

Wiring Up the Save Button

The edited record needs to be saved to the database. The following steps illustrate how to do this, using the generic `mergeEntity` method on the service facade.

1. In the Data Control palette, locate the `mergeEntity(Object)` method. Drag it to the Visual Editor and drop it on the Save button that you created earlier in this chapter. From the pop-up menu, choose **Bind Existing CommandButton**.
2. In the **Action Binding Editor**, click in the **Value** field, and then click the **[…]** that appears. In the **Variables** dialog box, expand the **ADF Bindings** node and then the **bindings** node. Locate and expand the **findServiceRequestById Iter** node and then the **currentRow** node. Select **dataProvider** and shuttle it across to the **Expression** pane. Click **OK**, and then click **OK** again.



3. In the Property Inspector, for the Save button, set the Action property to **`#{userState.retrieveReturnNavigationRule}`**. This method determines which page to return to after the changes have been saved.

4. As in the previous section, set an actionListener to fire when the Save button is clicked, and trigger a refresh. Right-click the **Save** button and, from the context menu, choose **Insert inside af:commandButton - #{res['srdemo.save']}→ADF Faces Core→SetActionListener**.

5. In the Insert SetActionListener dialog box, type **`#{true}`**in the From* field and **`#{userState.refresh}`** in the To* field. Click **OK**.

6. Save the page.

Disabling Input Fields for Closed Service Requests

When a service request has a status of “closed,” the problemDescription and assignedDate fields need to be disabled. The following steps show you how to do this:

1. Select the status list, and set the AutoSubmit and Immediate properties to **`true`**.

   Setting AutoSubmit to `true` for the status attribute causes a partial submit of the form when the status attribute is updated.

2. For both the problemDescription and assignedDate fields, set the Disabled property to **`#{backing_app_staff_SREdit.statusSelection.value=='2'}`**.

   To do this, click in the **Disabled** property in the Property Inspector, and then click the **Bind to Data** icon in the toolbar. In the Disabled dialog box, expand the **JSF Managed Beans** node and then the **backing_app_staff_SREdit** node. Scroll down to locate **statusSelection** and expand it. Scroll through the list to select **value**. Shuttle it across to the Expression pane. In the Expression pane, edit the expression to add the value **`2`**. Click **OK**.

   The enumeration list that you created in the List Binding Editor is zero index based, and Closed is the third entry; the value is therefore `2`.

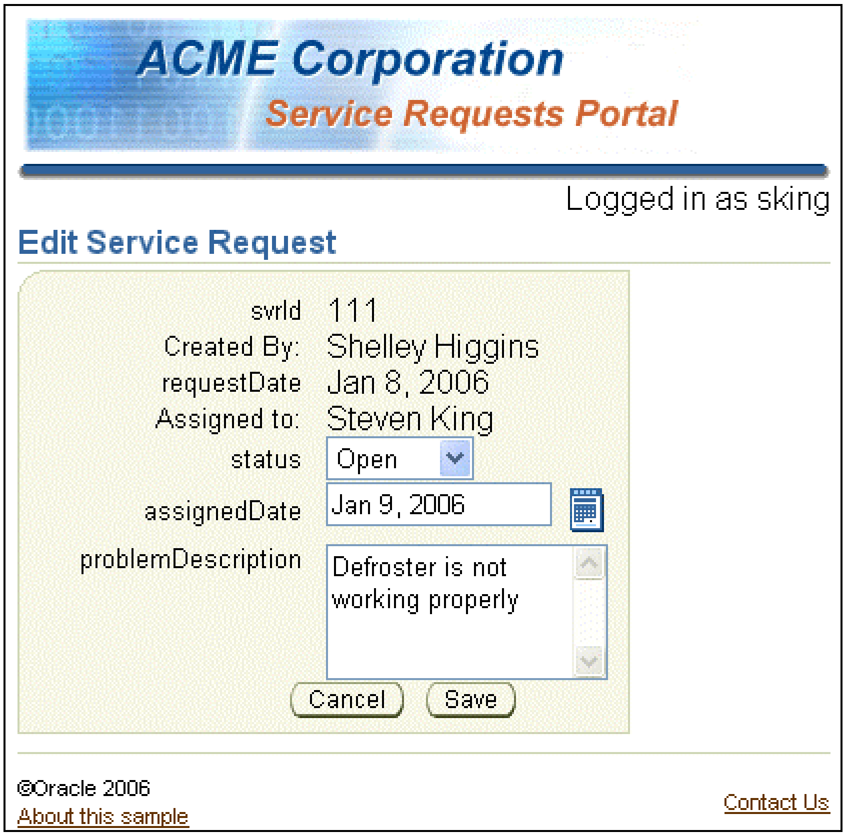
3. Set the PartialTriggers property for the two fields to **`statusSelection`**. As you saw earlier, this means that a change in the value of the status attribute causes the field to refresh.

4. Save the page.

Running the Page

The SREdit page receives the record to edit from the SRList page. Run the SRList page and, when prompted, use **`sking`** as the username and **`welcome`** as the password. Select the service request

for the id of `111` and click the **Edit** button. The SREdit page is displayed with the record ready for editing. The page should look something like the following screenshot.



1. On the page, test the drop-down list in the Status field.
2. Check that the date picker field works correctly.
3. Check that when you set the status of the request to Closed, the Problem Description and Assigned On fields are disabled.
4. Click the **Save** button and make sure that you return to the List page.
5. Make a change and then click **Cancel**. The change should be undone.

Changing the Application Look and Feel

You can change the look and feel of an application by changing its "skin." A skin is a global style sheet that needs to be set in only one place for the entire application. The application developer does not need to add any code, and any changes to the skin are picked up at run time. Skins are based on the Cascading Style Sheets specification.

By default, ADF Faces applications use the Oracle skin, and so this is the look and feel that has been applied to the pages you have created in the SRDemo application. ADF Faces also provides two other skins, the Minimal skin (which provides some basic formatting) and the Simple skin (which provides almost no special formatting). Alternatively, you can create a custom skin specifically for your application; to find out how to do this, refer to the JDeveloper Help pages.

Perform the following steps to apply the Minimal skin to your SRDemo application:

1. In the Navigator, locate the **UserInterface → WEB-INF → adf-faces-config.xml** file and double-click it to open it.
2. Change the <skin-family> value from **<skin-family>oracle</skin-family>** to **`<skin-family>minimal</skin-family>`**.
3. Save the file.
4. To see the new look and feel, run the application as **`bernst`**. Open the **SRList** page.

   Notice that the components in the Visual Editor also display the new style.
5. Run the **SRList** page. The page with the Minimal skin applied should look like the screenshot below. Notice that the buttons are squared off, the menu bar is simplified, and the labels are colored green.



Summary

In this chapter, you created an Edit page that enables managers and technicians to modify service requests. To accomplish this, you performed the following key tasks:

- Created a framework page based on the template page that you defined in Chapter 4
- Added some UI elements to the page to display the service request information
- Created lookups for the `createdBy` and `assignedTo` names
- Added a drop-down list for the `status` attribute
- Defined specific behaviors for the Save and Cancel buttons
- Disabled some input fields for requests with a status of Closed
- Changed the appearance of the application by applying a different skin

11

Deploying the Application to Oracle Application Server 10*g*

In this chapter, you use JDeveloper to create a deployable J2EE Web archive that contains your application and a few required deployment descriptors. You deploy the application to Oracle Application Server 10*g* using the JDeveloper deployment mechanism. You can then test the application and view its performance by using Oracle Enterprise Manager.

This chapter contains the following sections:

Introduction

Deploying a J2EE application is the last step in developing an application. After the application is completed and works as planned, the next step is to deploy it to a location where customers can use it.

JDeveloper has the built-in capability to deploy applications to a number of application servers. In this chapter, you deploy your application to Oracle Application Server 10*g*.

You perform the following key tasks in this chapter:

- **Create a connection to an existing Oracle Application Server 10*g*:** JDeveloper needs to establish a connection to the target application server so it can create the correct deployment profiles and push the completed files to the server.
- **Create deployment profiles:** These profiles manage the content and type of files required for deployment to the target application server.
- **Start Enterprise Manager (EM):** With EM, you can monitor or even redeploy an application.

  Note: If you do not have access to Oracle Application Server 10*g*, you will start an OC4J instance (this is an instance of OC4J that is not run from within JDeveloper).
- **Deploy the application:** After the profiles are created, you can deploy the application from within JDeveloper.
- **Test the deployment:** When the application is deployed, you can run it from a Web browser to verify that the application works as expected.
- **Use Enterprise Manager to explore the application server:** Enterprise Manager provides a detailed view of all the components of the application server. You can monitor and even change application parameters and fine-tune the performance of the application server.

Note: If you did not successfully complete Chapter 10, you can use the end-of-chapter application that is part of the tutorial setup.

1. Create a subdirectory named `Chapter11` to hold the starter application. If you used the default settings, it should be in `<jdev_install>\jdev\mywork\Chapter11`.
2. Unzip `<tutorial_setup>\starterApplications\SRDemo-EndOfChapter10.zip` into this new directory. Using a new separate directory keeps this starter application and your previous work separate.
3. In JDeveloper, close your version of the SRDemo application workspace.
4. Select **File > Open**, and then select `<jdev_install>\jdev\mywork\Chapter11\SRDemo\SRDemo.jws`. This opens the starter application for this chapter.

You can now continue with this tutorial using an application that implements all of the steps from Chapter 10.

Creating a Connection to Oracle Application Server 10*g*

JDeveloper supports deploying your applications to a variety of production application servers, via application sever connections. The first step in using JDeveloper to deploy an application is to create a connection to the target application server.

1. Click the **Connections** tab in JDeveloper.

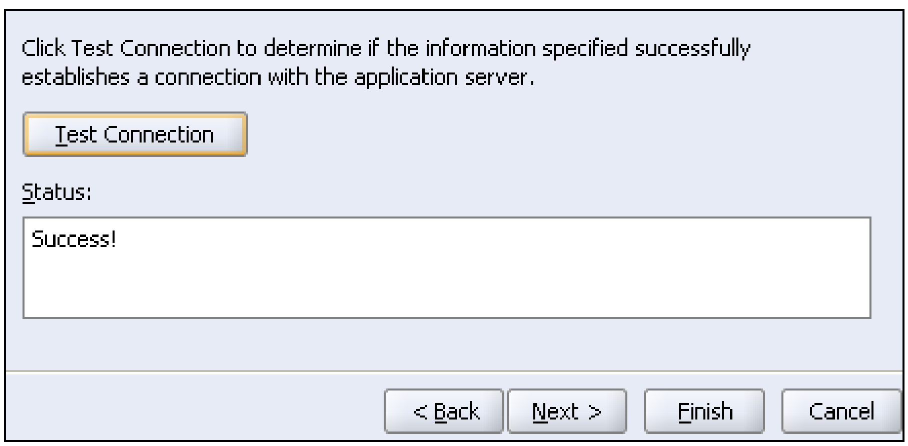
   If the Connections tab is not visible, you can select **View → Connections Manager** from the JDeveloper menu bar. (You can also press [Ctrl] + Shift + [O].)

2. Right-click **Application Server** in the Connections window and select **New Application Server Connection** from the context menu.

3. Enter the following values in the Create Application Server Connection Wizard:

Field	Value
Connection Name	`OracleAS10g`
Connection Type	`Oracle Application Server 10g 10.1.3`
Username	`oc4jadmin`
Password	Enter the administrator password for your instance.
Host Name	`localhost`

4. On the last page of the wizard, click **Test Connection**. You should see a success message.



5. When the test is successful, click **Finish** to create the connection.

Note: If you do not have access to Oracle Application Server 10*g*, follow the next sections to create an instance of OC4J and a JDeveloper connection.

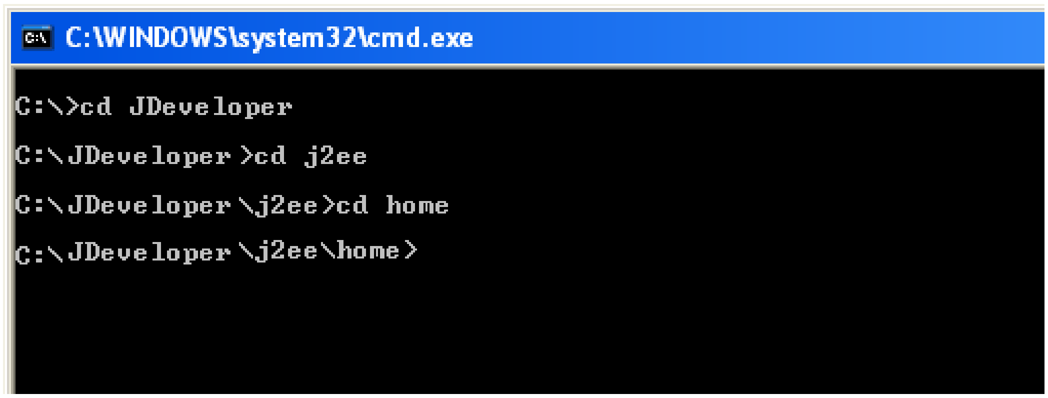
Starting an OracleAS Containers for J2EE (OC4J) Instance

JDeveloper includes an installation of OC4J and a JDK. In this section, you navigate to the directory where the `oc4j.jar` file is stored and start OC4J using the supplied version of the JDK.

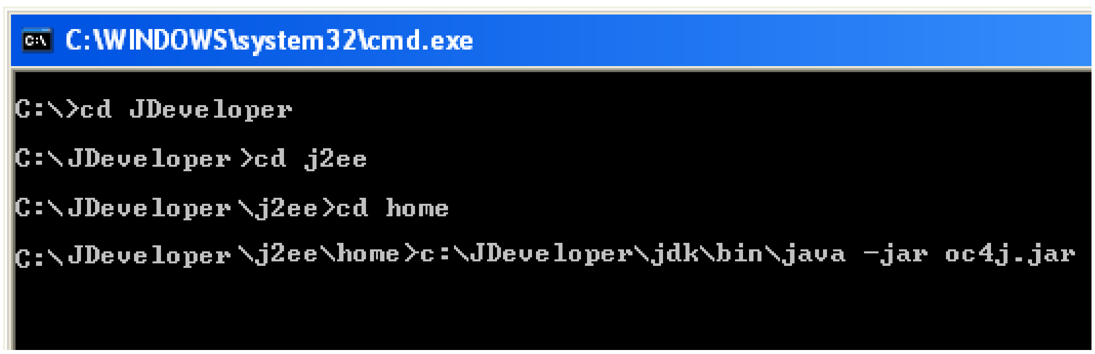
1. Open a Windows command window: Click **Start** on the Windows Start bar and select

Run.

2. Enter **cmd** in the dialog box and click **OK**.
3. In the command window, change the current directory to **<jdev_install>\j2ee\home**.



4. Use the supplied JDK to start OC4J by entering the following command:
<jdev_install>\jdk\bin\java -jar oc4j.jar



5. If this is the first time you have run OC4J, it automatically installs and prompts you for a password for the administrator account. The administrator account is `oc4jadmin`. Enter the password **admin1**. The installation asks you to confirm the password by entering it twice.
6. When the install and startup are complete, you will see the following message:
`Oracle Containers for J2EE 10g <10.1.3.1.0> initialized`

OC4J is now running and ready for use.

Creating a Connection to OC4J

The general steps to create a connection to any application server are basically the same. The differences come from the specific connection requirements of the server. The differences between creating a connection to Oracle Application Server 10*g* are fundamentally the same as creating a connection to OC4J. There are only a few differences in the arguments you supply.

1. Click the **Connections** tab in JDeveloper.

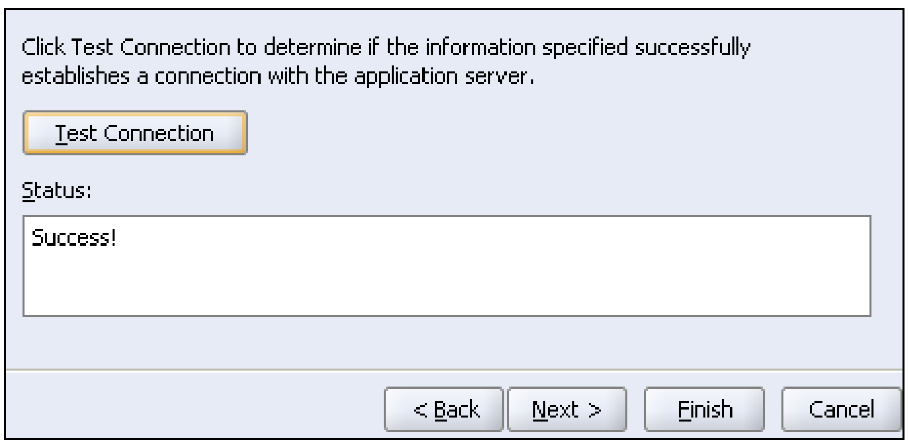
   If the Connections tab is not visible, you can select **View → Connections Manager** from

the JDeveloper menu bar. (You can also press **[Ctrl] + [Shift] + [O]**.)

2. Right-click **Application Server** in the Connections window and select **New Application Server Connection** from the context menu.
3. Enter the following values in the Create Application Server Connection Wizard:

Field	Value
Connection Name	`OC4J`
Connection Type	`Standalone OC4J 10g 10.1.3`
Username	`oc4jadmin`
Password	`admin1`
Host Name	`localhost`

4. On the last page of the wizard, click **Test Connection**. You should see a success message.



5. When the test is successful, click **Finish** to create the connection.

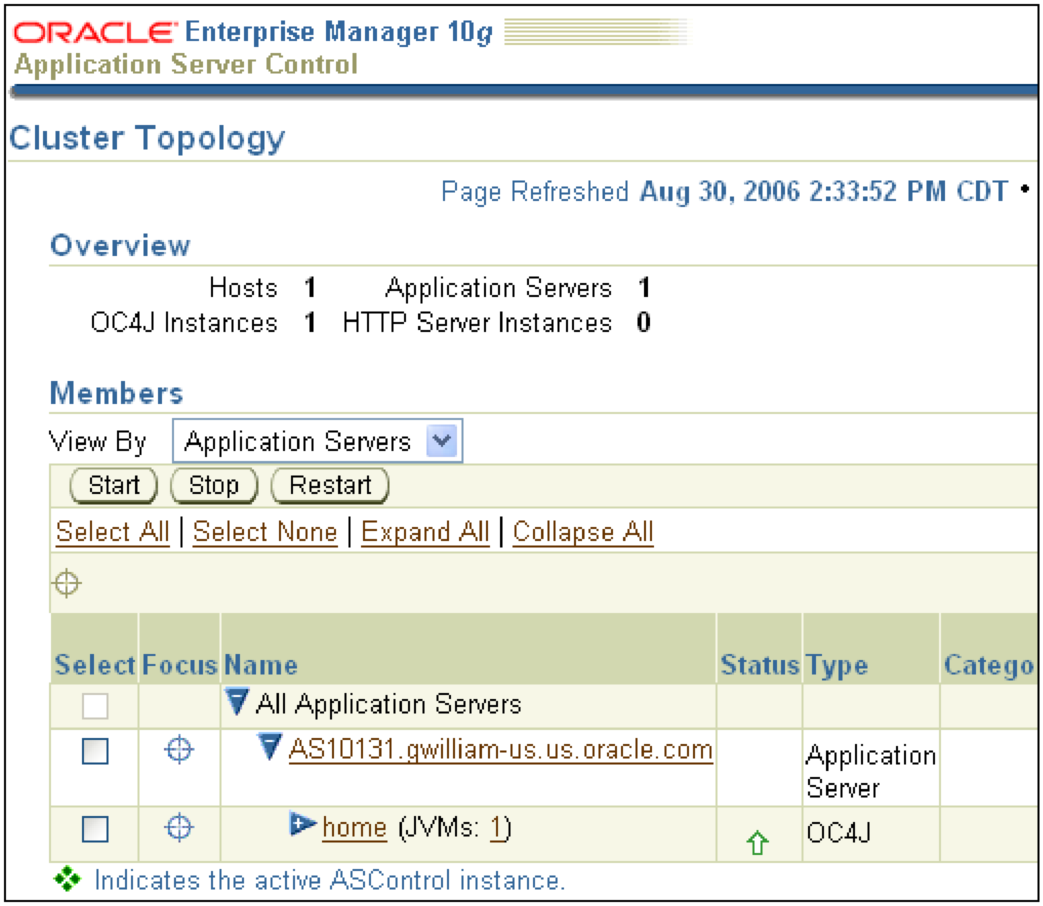
Starting Enterprise Manager

Oracle Application Server 10*g* comes with a browser-based Enterprise Manager (EM). Through this EM interface, you can monitor activities and applications deployed to the application server. After the application server is running, you can connect to EM using a browser. The next few steps open this interface and briefly explore the application server.

The stand-alone OC4J that comes with JDeveloper also includes Enterprise Manager.

1. Open a browser of your choice (Firefox, Internet Explorer, or another browser) and enter the following address: `http://127.0.0.1:8888/em`. If you are using stand-alone OC4J the address is the same.
2. Enterprise Manager 10*g* prompts you for a username and password. The username is **`oc4jadmin`** with a password of **`admin1`** (or your administrator password). Click Login to enter EM.

3. After successful login, the browser looks like the following:



You can now explore applications, Web services, and other components of Oracle Application Server 10*g*.

Preparing for Deployment

To keep all the elements of your application organized and cleanly separated, you will create a project to hold all of the deployment components for your application. This project will hold the deployment profiles and the deployment files (`.ear`, `.war`, `.jar`).

1. In the Applications Navigator, right-click **SRDemo** and select **New Project** from the context menu.
2. Select **Empty Project** in the New Gallery and click **OK**.
3. Name the project **`Deployment`**.

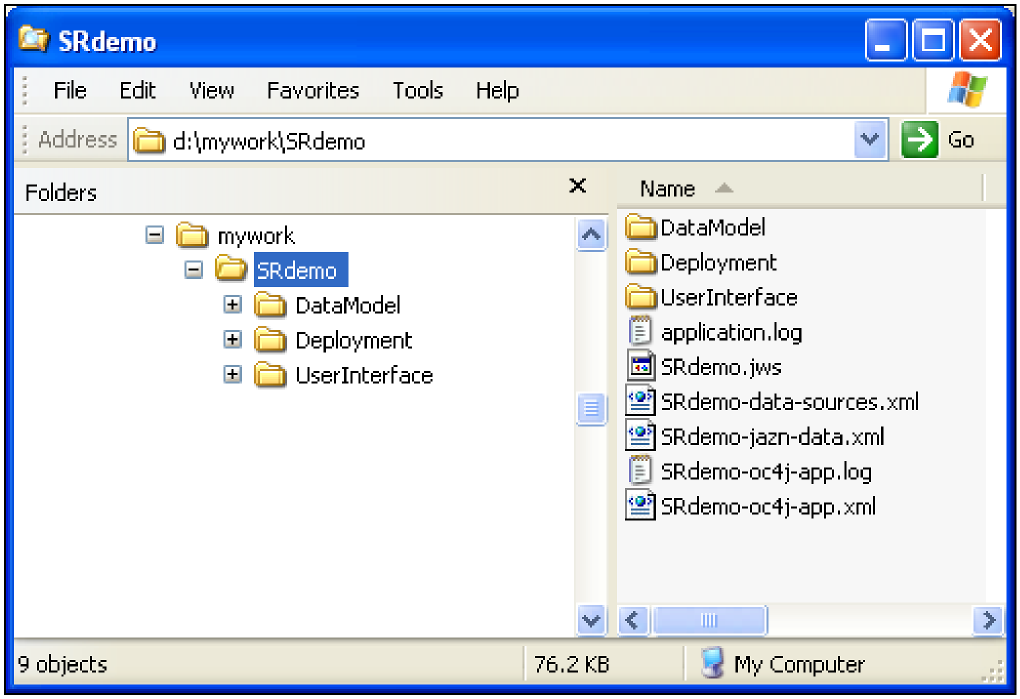
You now have a project to use to manage the deployment for your application.

When you were creating the Service Request application, you created users and roles in a file named `SRDemo-jazn-data.xml`. That is the default name that JDeveloper uses to segregate files by application. It places the files in a top-level directory by default.

That is useful for testing within the JDeveloper development environment. However, those

names and the directory structure are not directly deployable to a J2EE server. A J2EE server has some very specific requirements for names and locations of files that are to be deployed. In the next few steps, you move the files and rename them to standard J2EE deployment names.

1. Open Windows Explorer (or equivalent program on your operating system) and navigate to the root directory of your application.



2. Expand the Deployment node and create a subdirectory structure such as the following:

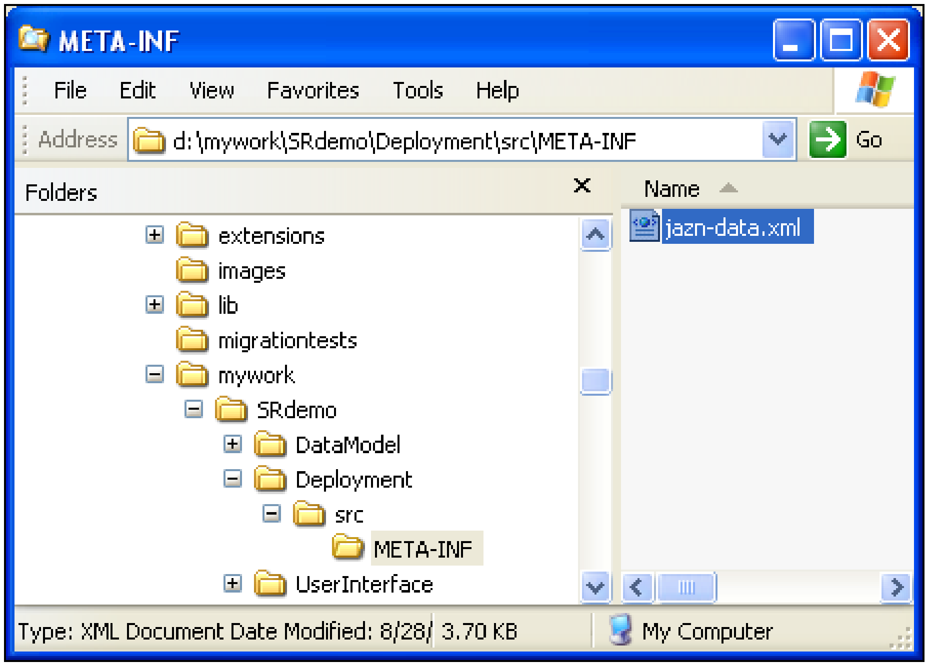
   `..\SRDemo\Deployment\src\META-INF`

3. This is the structure required by J2EE application servers for the `jazn-data.xml` and `data-sources.xml files`. Create a directory.

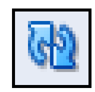
   If you're using Windows, you can select the parent in the left window, right-click in the right window, and then select New → folder from the context menu.

4. Copy the `SRDemo-jazn-data.xml` file to the new directory.

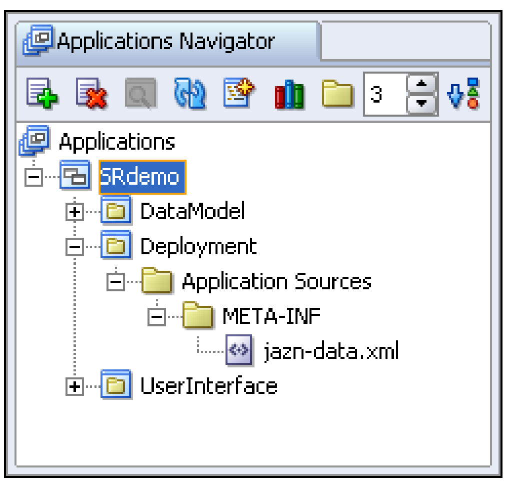
5. Rename the file to remove `SRDemo` from the name. The resulting file name should be `jazn-data.xml`. The directory structure should now look like the following screenshot:



6. Click the **Refresh** button in the Applications Navigator in JDeveloper.



7. The project structure should now appear as follows:



The file names and directory structure are now ready for deployment. The last thing you need to do before deployment is to make sure the deployment profiles use the TopLink transaction control. There are a couple of parameters at the session level that need to point to the correct transaction class.

In the next few steps, you modify the `sessions.xml` file to use the correct transaction management class.

1. In the Applications Navigator, select **DataModel→ Application Sources → TopLink**.
2. Select **sessions.xml**. In the Structure pane, double-click **SRDemoSession**.

3. On the **General** page, select the **External Transaction Controller** check box in **Options**.
4. Enter the following transaction class:

   `oracle.toplink.transaction.oc4j.Oc4jTransactionController`
5. Click the **Login** tab, and then click the **Options** tab.
6. Set the External Connection Pool to **True**.
7. Set the External Transaction Controller to **True**.
8. Select **Tools | Preferences** from the JDeveloper menu.
9. Click **Deployment**.
10. Clear the **Bundle Default data-sources.xml During Deployment** option.
11. Click **OK**.

The application is now ready to deploy.

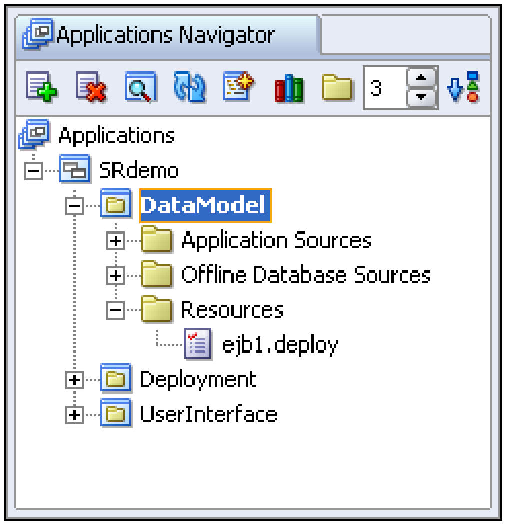
Creating Deployment Profiles

Deployment profiles are project components that manage the deployment of an application. A deployment profile lists the source files, deployment descriptors (as needed), and other auxiliary files that will be included in a deployment package.

There are three parts of the deployment package for the service request application: The DataModel project (`.jar`), the UserInterface project (`.war`), and the Deployment project (`.ear`.) files. You create deployment profiles for each of the three parts in this section of the tutorial.

The first deployment profile you create is for the DataModel project. The contents of this project are primarily the Java classes that make up the data model portion of the application. The deployment type for this project is an EJB JAR (Java Archive) file.

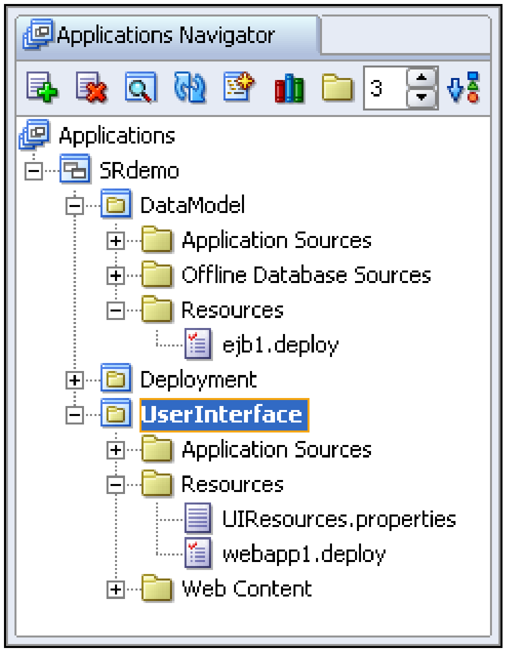
1. Right-click the **DataModel** project in the Applications Navigator and select **New** from the context menu.
2. Select **General →Deployment Profiles → EJB JAR File** in the New Gallery and click **OK**.
3. Click **OK** to accept the default profile name.
4. Click **OK** to accept the default values in the EJB JAR Deployment Profile Properties dialog box.
5. **Save** your work.
6. The Applications Navigator should now look like the following screenshot:



Creating the UserInterface Deployment Profile

You now create a deployment profile for the UserInterface project. This project is where you created the user interface components of the application. The deployment file for this project is a `.war` file (Web Archive, for the Web components).

1. Right-click the **UserInterface** project in the Applications Navigator, and select **New** from the context menu.
2. Select **Deployment Profiles → WAR File** in the New Gallery and click **OK**.
3. Click **OK** to accept the default profile name.
4. Change the Web Application's Context Root to **SRDemo**. This becomes part of the URL customers use to access the application.
5. Click **OK** to accept the default values in the WAR Deployment Profile Properties dialog box.
6. **Save** your work.
7. The Applications Navigator should now look like the following screenshot:



Creating the Archives

Now that you have created the deployment profiles, you can assemble the application into a deployable package.

In the assembly part of deployment, you create a deployment profile that includes any `.jar` and `.war` files you need for your application, along with other server configuration files that may be required. As you have already seen, a few of those files are the `data-sources.xml` and `jazn-data.xml` files.

1. Right-click the **Deployment** project in the Applications Navigator and select **New**.
2. Select **Deployment Profiles → EAR File** in the New Gallery and click OK**.**
3. Change the profile name to **SRDemoApplication** and click **OK**.
4. Click the **Application Assembly** category.
5. Select the following J2EE modules to include in the `.ear` file:

   UserInterface.jpr → webapp1.deploy

   Model.jpr → ejb1.deploy
6. Click **OK** to accept the other defaults.

You have now created the deployment profiles you need to successfully deploy your application to a J2EE server. In the next section, you deploy and test the application.

Creating a Connection Pool and Data Source

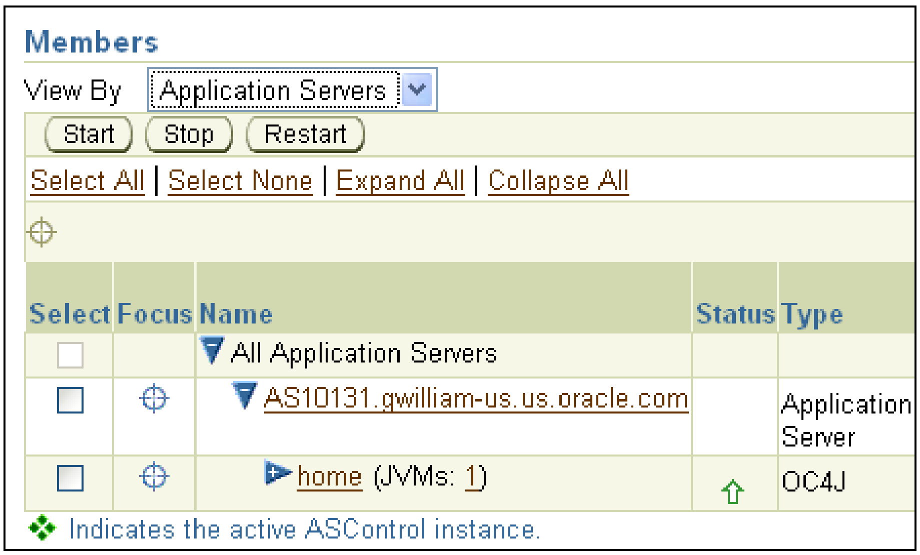
Applications deployed to Oracle Application Server 10*g* use a data source and connection pool to manage database access. When you are developing the application, JDeveloper creates a `data-sources.xml` file for you. It uses that file during development and internal testing. When it

comes time to deploy the application, you will create a Data Source directly on the application server.

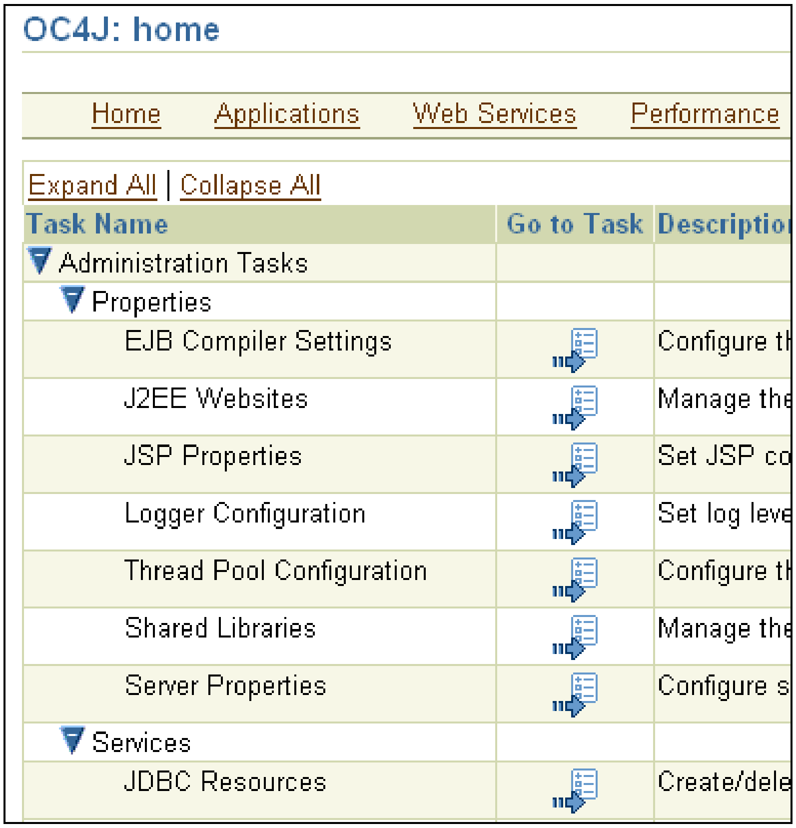
Creating the Connection Pool

In this section, you create a connect pool and a data source for your application. The name of the data source used by your application is SRDemoDS. The data source that you create using EM must be named the same as the data source that you used during development.

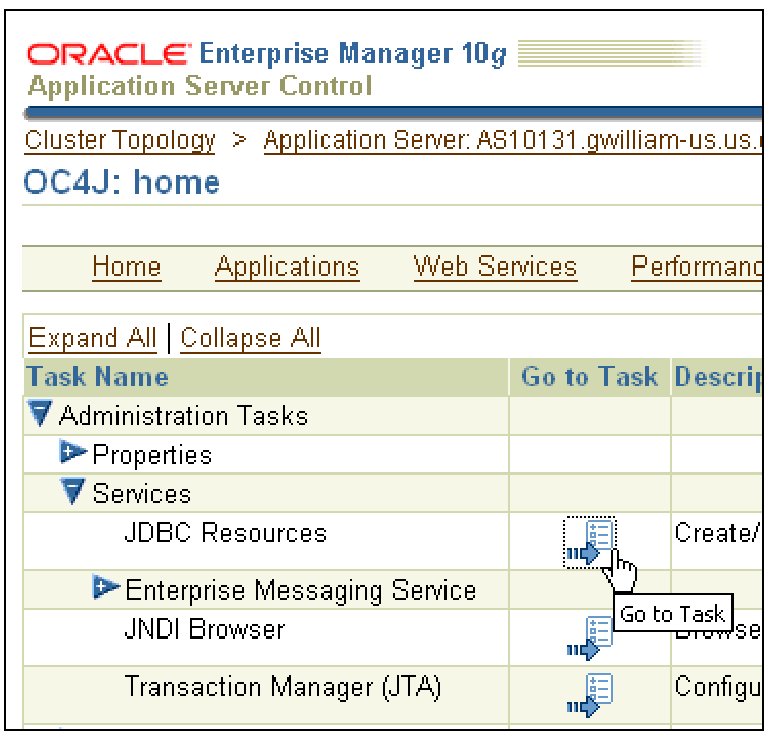
1. Open a browser and connect to Enterprise Manager (EM), if you haven't already done so.
2. The default opening page shows the Application Servers. If you need to change the view, select Application Servers in the View By drop list.



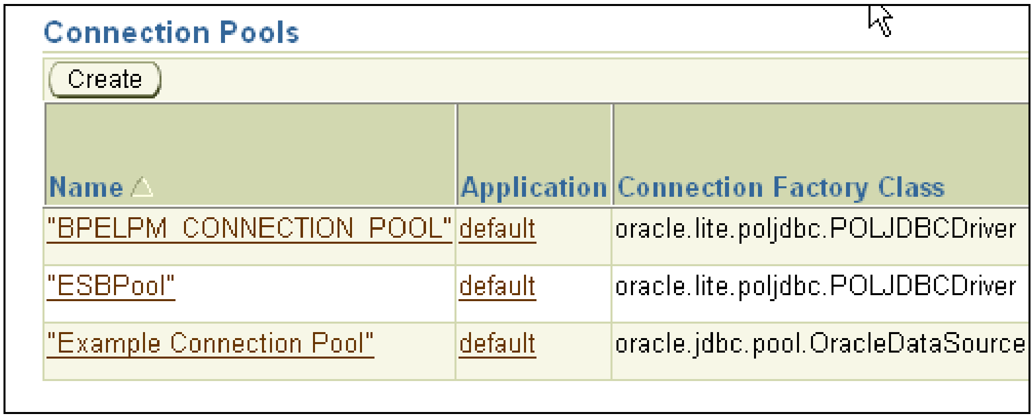
3. Click the **home** link to go to the **OC4J: home** page.
4. Click the **Administration** link.



5. Under Services, click **Go to Task** icon on the **JDBC Resources** line, which will show existing data sources and connection pools.



6. Click the **Create** button in the **Connection Pools** table.



7. Create the new Connection Pool using the following information.

Field	Value
Application	`Default`
Connection Pool Type	`New Connection Pool`

8. Click **Continue**.

Field	Value
Name	`SRDemoDSPool`
Connection Factory	`oracle.jdbc.pool.OracleDataSource` (the default value)
JDBC URL	`jdbc:oracle:thin:@//localhost:1521/orcl` (your database)
Credentials: Username	`srdemo`
Credentials: Cleartext Password	`oracle`

9. Click **Test Connection**. On the Test Connection page, click **Test**. If the connection is correct, the test page will close and you will see a success message at the top of the Create Connection Pool page. If the connection is not successful, correct the connection information and try again.
10. When it is successful, click **Finish** to create the connection pool.
11. Click **Test Connection**, which will take you to a test page. Click **Test**. You should see the following message in the Confirmation area at the top of the page.

```
Connection to SRDemoDSPool established successfully.
```

Creating the Data Source

Next, you create a data source that uses the SRDemoDSPool connection pool.

1. Click the **Create** button in the Data Sources table.

2. Use the following information to create the data source:

Field	Value
Application	`Default`
Data Source Type	`Managed Data Source`

3. Click **Continue**.

Field	Value
Name	`SRDemoDS`
JNDI Location	`jdbc/SRDemoDS`
Transaction Level	`Global & Local Transactions (default)`
Connection Pool	`SRDemoDSPool`
Login Timeout	`0` (default)

4. Click **Finish**.
5. Click **Test Connection** for the SRDemoDS data source
6. On the Test Connection page, click the **Test** button.
7. You should see the following message:

   `Connection to "SRDemoDS" established successfully.`

Deploying the Application

JDeveloper provides a one-click option to deploy an application to an application server. After you have assembled the application into an EAR file, you can right-click the deployment profile and select the target application server.

1. In the Applications Navigator, select Deployment → Resources. Right-click the **`SRDemoApplication.deploy`** deployment profile.
2. Select **Deploy to → OracleAS10g** from the context menu. Remember that OracleAS10g is the name of the connection to Oracle Application Server 10*g* you created earlier in this tutorial.

   > **Note:** If you did not have access to Oracle Application Server 10*g* and used OC4J, use the name of the connection you created for OC4J, which should be **OC4J**.

   During deployment, JDeveloper re-creates the `.jar` and `.war` files and then assembles the `.ear` file. After the file is assembled, JDeveloper deploys the file and unpacks it in a directory on the application server, depending on the target environment.
3. Click **OK** to accept the Application Configuration and begin the deployment.
4. When the deployment is complete, the following messages appear in the JDeveloper log

window:

```
Committing ClassLoader(s)
Initialize webappl begins...
Initialize webappl ends...
Started application : SRDemoApplication
Binding web application(s) to site default-web-site begins...
Binding webappl web-module for application SRDemoApplication to site
Initializing Servlet: javax.faces.webapp.FacesServlet for web applica
Binding web application(s) to site default-web-site ends...
Application Deployer for SRDemoApplication COMPLETES. Operation time:
Elapsed time for deployment:  54 seconds
----  Deployment finished.  ----    Aug 31, 2006 9:57:51 AM
```

Testing the Application

You have now deployed the application to a stand-alone instance of OC4J. You can test the application with a Web browser using the context root that you specified for the application.

1. Open a Web browser.
2. Enter the URL **`http://localhost:8888/SRDemo/faces/app/SRList.jspx`**.
3. This directs you to the logon page of the application. Use the following as login credentials:

Field	Value
Username	`sking`
Password	`welcome`

4. When you have explored the application, leave the browser open.

Using Enterprise Manager to Explore the Application Server

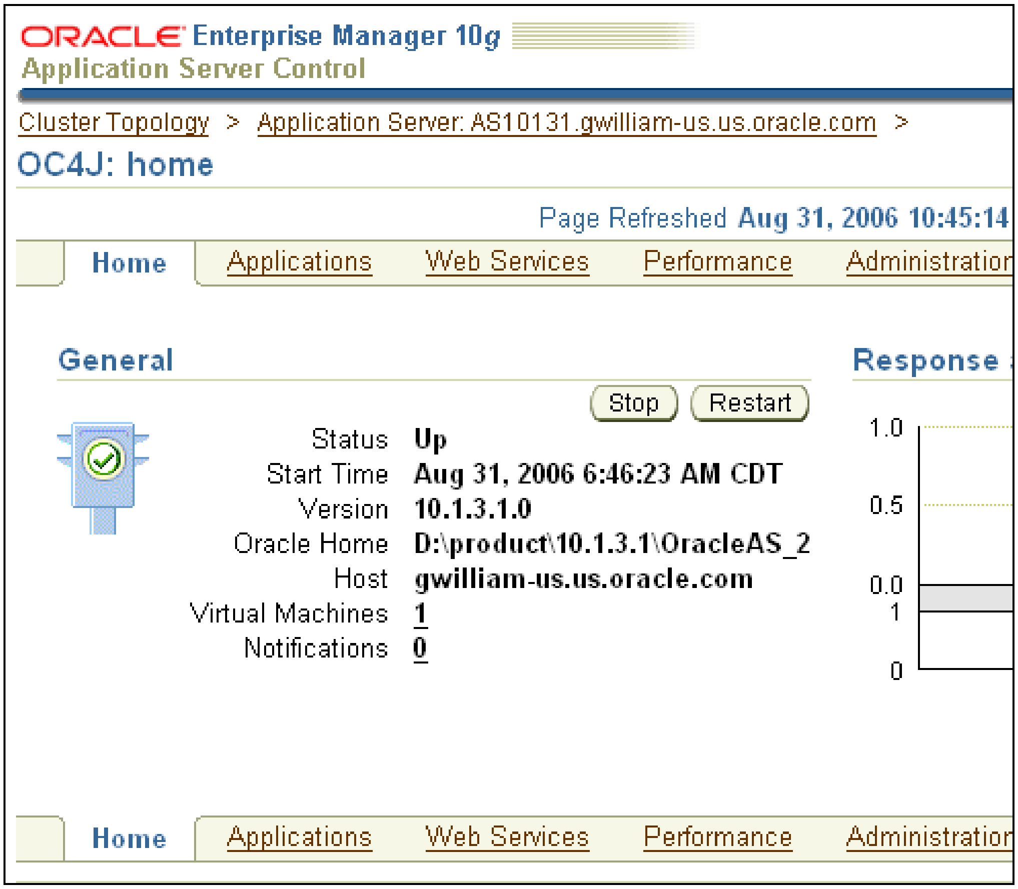
OC4J is delivered with a Web-based Enterprise Manager (EM) that you can use to monitor and manage deployed applications. In this final section, you use EM to explore your application and the application server.

1. Open a second Web browser.
2. Enter the URL **`http://localhost:8888/em.`**
3. Use the following credentials to log in to EM:

Field	Value
Username	`oc4jadmin`
Password	Enter the administrator password for your instance.

4. After you log in, you are directed to the EM home page, which should look like the

following screenshot:



You can explore a number of aspects of the application using EM, such as the following:

5. Click the **Applications** tab, and then click **SRDemoApplication** to view specifics about the application you just deployed.
6. Click the **Performance** link to see graphs of application performance.
7. Click the **Administration** link to see all of the deployment aspects of the application, including details of the JAZN security implementation.

Summary

In this chapter, you saw how to use JDeveloper to deploy an application to OC4J. JDeveloper enables you to easily deploy an application to a number of different application servers by using deployment profiles and application server connections.

Enterprise Manager is an easy-to-use and powerful interface for the management of Oracle Application Server and deployed applications.

Here are the key tasks that you performed in this chapter:

- Created a connection to an existing Oracle Application Server 10*g*
- Created deployment profiles

- Started Enterprise Manager (EM)
- Deployed the application
- Tested the deployment
- Used Enterprise Manager to explore the application server